NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS COMPLETION APPROVAL FORM

THESIS TITLE: Scaling Contracts to Realistic Languages

AUTHOR: T. Stephen Strickland

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor Date

Thesis Reader

Thesis Reader

Thesis Reader

GRADUATE SCHOOL APPROVAL:

Director, Graduate School

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature

Distribution: One Copy to Thesis Advisor
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)
Scaling Contracts to Realistic Languages

A dissertation presented
by

T. Stephen Strickland

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts
July, 2012
Abstract

Contracts allow programmers to specify the expected behavior and use of program components separately from the code of the components themselves. Since Bertrand Meyer introduced contracts to working programmers via the Eiffel programming language, Eiffel-like contract systems have been designed for many other object-oriented languages. Contract systems are not limited to object-oriented programming; Findler and Felleisen showed how to add contracts to languages with higher-order functions and formalized the notions of contract boundaries and blame.

Currently, contract systems come with two major omissions: monitoring the invariants of mutable data structures and protecting first-class components, which are used in the construction of large-scale software projects. This dissertation presents the design and implementation of contract systems that cover these language features. These contracts are expressive enough to describe the protection of these values and efficient enough that performance concerns do not preclude their use.
# Contents

Abstract i

Contents iii

List of Figures vii

1 Programming in the Large 1

2 A History of Contracts 5
   2.1 Contracts in Eiffel 5
   2.2 Contracts in Racket 6
      2.2.1 Contract Boundaries 7
      2.2.2 Blame Tracking 8
      2.2.3 Contracts for Higher-Order Values 8

3 Contracts for First-Class Modules 11
   3.1 Units 11
      3.1.1 The Core Language 11
      3.1.2 ...plus Units 12
      3.1.3 Dynamic Semantics 14
      3.1.4 Unit Pragmatics 17
   3.2 Contracts 18
      3.2.1 Units with Contracts 19
      3.2.2 Adding Contracts Gradually 20
      3.2.3 A Contract Combinator for Units 20
   3.3 Units with Contracts 21
      3.3.1 Syntax for Units with Contracts 21
      3.3.2 The Idea 22
      3.3.3 Contract Compilation 23
      3.3.4 Dynamic Contract Checking 25
## CONTENTS

### 3.4 Implementation
- 3.4.1 Implementing Units ............................................. 30
- 3.4.2 Implementing Unit Contracts ................................... 31
- 3.4.3 Implementing unit/c ............................................. 31

### 3.5 Contracts for Nominally Linked Units
- 3.5.1 Signatures and Contracts ....................................... 33
- 3.5.2 Units without Contracts ........................................ 33
- 3.5.3 Units with Contracts ............................................ 34
- 3.5.4 Implementing Units as Contract Boundaries ................... 35

### 4 Contracts for First-Class Classes

#### 4.1 Designing Class Contracts ..................................... 37
- 4.1.1 Contracts on Classes ........................................... 40
- 4.1.2 Contracts, Blame, and Dynamic Dispatch .................... 42

#### 4.2 The Model ...................................................... 43
- 4.2.1 Contract PCF .................................................. 44
- 4.2.2 Adding First-Class Classes to CPCF ......................... 45
- 4.2.3 Well-Formed Programs ......................................... 46
- 4.2.4 CFCC Semantics ............................................... 48
- 4.2.5 CFCC Object-Oriented Semantics .............................. 50
- 4.2.6 CFFC Contract Semantics .................................... 51
- 4.2.7 CFCC Ownership Semantics .................................. 53

#### 4.3 Blame Correctness ............................................... 54

#### 4.4 Implementation ................................................. 55
- 4.4.1 Basics .......................................................... 56
- 4.4.2 Internal Dynamic Dispatch ................................... 58

#### 4.5 Evaluating Class Contracts ................................... 61
- 4.5.1 Evaluating the Performance Impact on Existing Code ...... 61
- 4.5.2 Evaluating Class Contract Features ......................... 63
- 4.5.3 Performance Evaluation ....................................... 66

#### 4.6 Interface Contracts ............................................. 68
- 4.6.1 Contracts on Interfaces ....................................... 68
- 4.6.2 Converting Interface Contracts to Projections .............. 70
- 4.6.3 Implementing Interface Contracts ............................. 72

### 5 Contracts for Mutable Values ..................................... 75

#### 5.1 Extensibility versus Reasoning ................................. 75
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Sample code from a DrRacket plugin</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>An Eiffel class for a stack of prime numbers</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>A Racket module for a stack of prime numbers</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Syntax for the base model</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Syntactic extensions for units</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Values and evaluation contexts for units</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>The semantics of the core language</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>Reduction rules for units</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>Linking clients to many servers</td>
<td>17</td>
</tr>
<tr>
<td>3.7</td>
<td>Surface syntax extensions for contracts</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>Extended example with predicate exports</td>
<td>22</td>
</tr>
<tr>
<td>3.9</td>
<td>Internal syntax for guard expressions</td>
<td>23</td>
</tr>
<tr>
<td>3.10</td>
<td>Contract compilation</td>
<td>24</td>
</tr>
<tr>
<td>3.11</td>
<td>New and revised reduction rules</td>
<td>26</td>
</tr>
<tr>
<td>3.12</td>
<td>Substitution and Definition Renaming</td>
<td>27</td>
</tr>
<tr>
<td>3.13</td>
<td>Signatures with contracts</td>
<td>32</td>
</tr>
<tr>
<td>3.14</td>
<td>Example interactive animation units</td>
<td>34</td>
</tr>
<tr>
<td>3.15</td>
<td>Mismatched signatures and contracts</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>CPCF syntax</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>CPCF syntax with Ownership and Obligations</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>CFCC syntax</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Ownership coincides with contract monitors</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Obligations coincide with labels on monitors</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Parameterized evaluation contexts</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Reduction semantics for CFCC</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>The $pull$ meta-function</td>
<td>51</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.9 The Find meta-function ........................................ 52
4.10 The Push meta-function ........................................ 52
4.11 Classes, objects, and contract boundaries ...................... 56
4.12 Method table organization for contracts ......................... 57
4.13 Second revision of class method tables for contracts .......... 58
4.14 Classes with override and inherit contracts .................... 59
4.15 Micro-benchmark results (times in milliseconds) ............... 62
4.16 Space benchmarks for classes .................................. 63
4.17 Classes, Interfaces, and Mixin Counts ........................ 64
4.18 Use of Contract Forms ........................................... 65
4.19 Contract performance (times in milliseconds) ................... 66
4.20 Frequency of contract execution in test components .......... 67
4.21 A complex interface hierarchy .................................. 71
4.22 Micro-benchmark results (times in milliseconds) ............... 73

5.1 Partial chaperone and impersonator API ......................... 87
5.2 VectorRacket Surface (left) and Machine State (right) Syntax .... 92
5.3 VectorRacket Evaluator (function clauses in order) .......... 92
5.4 VectorRacket Reductions ......................................... 94
5.5 VectorRacket Chaperone Syntax Extensions ...................... 94
5.6 VectorRacket Chaperone Reductions ............................. 95
5.7 VectorRacket Chaperone Metafunctions ........................ 96
5.8 Procedure-call microbenchmark results ......................... 99
5.9 Realistic procedure benchmark results ........................ 102
5.10 Vector microbenchmark results .................................. 103
5.11 Realistic vector benchmark results ............................ 103
5.12 Structure microbenchmark results .............................. 104
5.13 Realistic structure benchmark results ......................... 105

A.1 Ownership coincides with contract monitors (2) ................ 114
A.2 Ownership is preserved for monitored expressions ............. 114
Chapter 1

Programming in the Large

Modern programming languages provide the programmer with the expressive power to develop first-class components—modules and classes—that can be created and combined at runtime. Figure 1.1 contains sample code from the Racket [Flatt and PLT, 2010] code base that illustrates the use of first-class modules and classes. This code is the starting point for the macro debugger [Culpepper and Felleisen, 2007], which is a tool [Findler and PLT, 2010], or plugin, for the DrRacket development environment [Findler et al., 2002].

Figure 1.1 Sample code from a DrRacket plugin

```scheme
(define tool@
  (unit (import drracket:toolˆ) (export drracket:tool-exportsˆ)
    (define (phase1) ...)
    (define (phase2) (void))
  ...
    (define (macro-debugger-definitions-text-mixin %) (class % ...))
    (define (macro-debugger-interactions-text-mixin %) (class % ...))
  ...
    (drracket:get/extend:extend-interactions-text
     macro-debugger-interactions-text-mixin)
    (drracket:get/extend:extend-definitions-text
     macro-debugger-definitions-text-mixin) ...))
```

A DrRacket tool must define a unit [Flatt and Felleisen, 1998], which is a first-class module. Each unit has a list of imported and exported signatures. A signature is a collection of names. A tool unit must import the signature drracket:toolˆ, which lists the functions a tool can use to gain access to parts of DrRacket. The tool unit must also export drracket:tool-exportsˆ, which consists of two names, phase1 and phase2. These identifiers should be bound to thunks, and the bound thunks are invoked at appropriate points in the DrRacket loading process.

Finally, the macro debugger also defines two mixins [Bracha and Cook, 1990]. Since classes in Racket are first-class values [Flatt et al., 2006], mixins are functions that take
a class and return a subclass that provides the extended functionality. Each of the macro debugger's mixins extends an editor in the DrRacket GUI: one extends the editor for definitions, where the programmer writes his code, and the other extends the editor for interactions, which contains a read-eval-print loop for exploring the definitions. The macro debugger hooks each mixin into DrRacket through functions imported via the \texttt{drracket:tool} signature.

The example in figure 1.1 is in no way unique. Large programs written in Racket often rely on units and first-class classes. These components are first-class values with potentially unknown and dynamically determined linking contexts, and the programmer of a given component should be able to restrict imports and protect exports with behavioral software contracts.

Racket's contract system should support both first-class modules and first-class classes. The addition of appropriate contracts is challenging, however, due to the dynamic nature of these components. The contract system cannot determine which parties enter into these contracts until the components are linked, whereas in all existing contract systems both parties are known at compile-time.

In addition to first-class components, almost all modern programming languages contain a rich library of mutable data structures such as reference cells, vectors, and hash tables. The existing contract system covers these values, but those contracts check only first-order properties at contract boundaries. Hence, these contracts do not check that future mutation adheres to the contract, and so the properties suggested by the applied contracts can be broken without triggering a contract error.\footnote{Systems like Typed Racket [Tobin-Hochstadt and Felleisen, 2008] that depends on contract checking for type soundness are unsound with these contracts.} Also, it is not clear how to correctly apportion blame for uses of a mutable value. While there is existing work on contracts for object systems where objects contain mutable fields, these contracts follow the Eiffel convention, where contracts are checked only on entering and exiting methods. No attempt is made to detect violations immediately on field access or mutation and provide appropriate blame.

These omissions in existing contract systems can and should be rectified, which brings me to my thesis:

\textit{Higher-order contracts can provide expressive, efficient blame for first-class components and mutable values.}

To support this thesis, I have conducted research on the design, implementation, and performance of contracts for mutable values and first-class components. The addition of contracts may affect the performance of existing code, because support for contracts requires cooperation from the subsystems that provide these language features. Thus, an implementation
is necessary to ensure both that their inclusion does not unduly impact uncontracted code and that these contracts are usable in practice.

I begin with a brief history of the contract systems that form the foundation of my proposed work. Chapter 3 illustrates my design for a contract system that protects both structurally- and nominally-linked first-class module systems. Chapter 4 combines the two approaches from the previous chapters to present a system for protecting an object-oriented system that contains both first-class classes and objects. Chapter 5 describes the challenges posed by contracts for mutable values and presents a solution using two types of proxies.
Chapter 2

A History of Contracts

Parnas [1972] first suggested using specifications that describe the expected behavior and use of program units to control the interaction between separate modules. Meyer [1992a] popularized their use and coined the term “contract” as well as the phrase “design by contract.” In this chapter, we start with a description of Meyer’s contract system as implemented in Eiffel [Meyer, 1992b]. We then switch to the Racket contract system and elaborate on those attributes that shape our design of contracts for first-class components and mutable values: contract boundaries, blame tracking, and contracts for higher-order values.

2.1 Contracts in Eiffel

Eiffel supports three kinds of class contracts: method preconditions, method postconditions, and class invariants. Method preconditions specify the conditions that must hold when a method is called and are preceded by the require keyword. Method postconditions specify the conditions that must hold when a method returns and are preceded by the ensure keyword. Class invariants describe conditions that must hold on every method entry and exit and are written at the class top-level in a block that begins with the keyword invariant. All contract features contain a sequence of boolean expressions, each tagged with a label that is used when reporting contract violations.

Figure 2.1 sketches a class PRIME_STACK that implements a stack of prime numbers. It assumes a predicate isPrime on integers and a class LIST that implements a mutable list with isEmpty, add and remove operations. Both add and remove operate on the same end of the list. The class contains a private field intlist, in which the stack stores the items, and an invariant on intlist that requires it never be the value Void, which is the equivalent of Java’s null. The class also provides three public operations: push, pop, and isEmpty. The push operation requires that its argument be a prime integer. The pop operation requires that intlist is not empty and guarantees that the result is a prime number.
Checking all class invariants on both method entry and exit can lead to problems due to re-entrance [Szyplerski, 1997, p.66]. Any method that needs to temporarily break an invariant must perform all the actions necessary to restore the invariant locally. Recent object-oriented languages with contracts, such as Spec# [Barnett et al., 2004], provide the programmer with a mechanism to state that during the execution of a particular block of code, invariants may be broken. After the block is executed, the invariants are checked to ensure that they have been properly re-established. Racket solves this problem using contract boundaries.

2.2 Contracts in Racket

Figure 2.2 illustrates the use of Findler and Felleisen [2002]’s contracts in Racket. It contains code that implements the above Eiffel class as a stateful, procedural module. It also adds two additional procedures, which are used to explain the differences between Racket’s and Eiffel’s contract systems. The prefix → operator creates contracts for procedures. The module, dubbed prime-stack, assumes the existence of a predicate on numbers called prime?. It provides push and pop operations on a stack of prime numbers. When a client imports the

---

**Figure 2.1** An Eiffel class for a stack of prime numbers

```plaintext
class PRIME_STACK
feature
  isEmpty : BOOLEAN is
    do
      Result := intlist.isEmpty
    end
push(new_item : INTEGER) is
  require
    item_is_prime: isPrime(new_item)
  do
    intlist.add(new_item)
  end
pop : INTEGER is
  require
    stack_is_nonempty : isEmpty = False
  do
    Result := intlist.remove
  ensure
    isPrime(Result)
  end
feature {NONE}
  intlist : LIST
invariant
  intlist_not_void : intlist /= Void
end
```
2.2. CONTRACTS IN RACKET

Figure 2.2 A Racket module for a stack of prime numbers

```racket
#lang racket ;; name: prime-stack

(define intlist null)

(define (empty? ) (null? intlist))

(define (push i) (set! intlist (cons i intlist)))

(define (pop)
  (begin0 (car intlist)
    (set! intlist (cdr intlist))))

(define (add-and-remove n) (push n) (pop))

(define (map-prime f) (set! intlist (map f intlist)))

(provide/contract [empty?  (→ boolean?)]
  [push  (→ prime? void?)]
  [pop   (→ #:pre (not (empty?)) prime?)]
  [add-and-remove (→ number? number?)]
  [map-prime  (→ (→ prime? prime?) void?)])
```

The `prime-stack` module and uses its functionality, the argument to `push` and the result from `pop` are checked for primality. Additional preconditions can be specified with the `#:pre` keyword, as in the contract of `pop`, where the additional precondition specifies that the stack must be non-empty when `pop` is called.

2.2.1 Contract Boundaries

Contracts in Racket are checked only when values cross the module boundary. For our running example, a boundary crossing occurs when a value flows into or out of the `prime-stack` module via function calls. In constrast, calls internal to the module are not checked.

Consider the `add-and-remove` function. It first adds, then removes an arbitrary number from the stack. Since its calls to `push` and `pop` are internal to the `prime-stack` module, these calls are not checked and therefore cannot signal a contract violation. Hence, it is acceptable to call the function with a composite number such as 4.

The Racket experience shows that from a run-time verification perspective, contract checking at boundaries is as effective as contract checking for all method calls. First, programmers tend to write small modules or classes. Within these components, they tend to trust their method and function calls. Even if a component breaks a contract, the guilty component tends to be small, effectively restricting the search for errors. For large components, Racket provides `define/contract` and `with-contract` [Strickland and Felleisen,
2010b], i.e., constructs for breaking down a module into nested contract regions. With these, a programmer can control every unit of code, regardless of its size. Second, the boundary-oriented monitoring mechanism ensures that contract checking does not interfere with the proper implementation of tail calls. We consider the latter a critical element of proper program design, especially for object-oriented programming in the spirit of the popular design patterns.

2.2.2 Blame Tracking

Every module that imports `prime-stack` enters into an agreement with the `prime-stack` module. If a contract violation occurs, the appropriate contract party is blamed. For example, if a client calls the `push` function with a composite number, the contract monitoring system blames the client. If the client calls the `pop` function and a composite number were returned, a contract violation would be signaled that blames `prime-stack`.

When we discuss blame, we call the server the positive party and the client the negative party. These terms are analogous to the uses of the terms positive and negative to describe positions in implications in logic or function types in type systems.

2.2.3 Contracts for Higher-Order Values

Since Racket is a functional language, functions are first-class values. Of course, it is impossible to check contracts on functions, say `(→ prime? prime?)`, at the point where the function crosses the contract boundary. Instead, the contract system must delay checking until the function is applied. Then the contract system can check that the contracts on the arguments hold, and that the results satisfy their contracts too. Wrapper objects are the natural way to implement this form of checking. The key is to equip the wrapper with enough information so that the contract system can blame the guilty party in case of contract violations.

As seen with `push` and `pop`, the negative party is responsible for the contracts on the arguments to exported functions and the positive party is responsible for the contracts on values returned from exported functions. The reasoning behind this becomes clear when we consider functions as opening channels across the contract boundary, where arguments flow from the client module to the server and the results from the server module to the client.

This reasoning generalizes to cases where the arguments provided by the client are functions, as in the argument `f` to the `map-prime` function. The arguments to those functions originate in the server, i.e., the positive party, and thus it is responsible for these arguments of arguments, while the negative party is responsible for the result. Indeed, the roles are
swapped again if any of the doubly-nested arguments are functions. So higher-order function contracts require swapping the positive and negative parties for each nested function contract.
Chapter 3

Contracts for First-class Modules

In this chapter, we explore two different contract systems for first-class modules. In sections 3.1–3.4, we explain the design and implementation of a contract system for structurally-linked first-class modules. This contract system adds contracts to the listed imports and exports of individual modules as well as a contract combinator for contracting already existing modules. In section 3.5, we examine how to adapt such a system to nominally-linked first-class modules. This system adds contracts to the signatures that describe interfaces shared between nominally-linked modules.

3.1 Units

Flatt and Felleisen [1998] introduced units as a dynamic and dynamically-typed analog to SML’s functors. Unlike the latter, units are first-class values that programs can manipulate at run-time (compound, invoke, splice). Furthermore, while functors act as transformations on atomic modules, linking specifications for units may arrange them in potentially cyclical graphs. Last but not least, programs may decide at run-time which units to link in, and indeed, they may even load a unit from a file after the execution has started.

The original model of units comes with both a reduction semantics and a type system. In this section, we acquaint the reader with units and their untyped operational model. This section ignores types because they are orthogonal to a study of contracts.

3.1.1 The Core Language

Following tradition [Leroy, 1994], the original unit model is parameterized over the core language, making only some minimal assumptions such as the existence of mutually recursive definitions.

---

1 Prior versions of this work appeared at IFL 2009 [Strickland and Felleisen, 2010b] and DLS 2009 [Strickland and Felleisen, 2009].
For concreteness, we start from the higher-order functional core language shown in figure 3.1. A program in this language is a list of definitions followed by an expression. The top-level definitions define distinct variables.

Figure 3.1 Syntax for the base model

\[
\begin{align*}
p &::= \text{(program } d \ldots e) \\
d &::= \text{(define } x e) \\
e &::= x \mid n \mid b \mid x \mid (\lambda (x) e) \mid (e e) \mid (op e e) \mid (\text{if } e e) \\
op &::= + \mid - \mid * \mid = \mid <= \\
b &::= \#\mid \#f
\end{align*}
\]

In addition to functions as first-class values, the language includes three kinds of atomic values:

- numbers \((n)\),
- strings \((s)\),
- and booleans \((b)\) using Scheme notation.

It also supports operations on numbers and conditional expressions.

While Racket includes assignment statements, destructive structure operations, and powerful control operations, our model omits such imperative extensions because their inclusion would only increase the notational overhead without any additional benefits. As Flatt and Felleisen [1998] explained, units are entirely orthogonal to the existence of imperative features in a language.

### 3.1.2 ... plus Units

To add units to our model language, we extend the syntax with the additional terms in figure 3.2. The **unit** form creates an atomic component, which is a first-class value. A **compound-unit** operation synthesizes a unit hierarchy from two existing units, usually resolving some imports in the process. Finally, the **invoke-unit** operation evaluates the body of a unit that requires no more imports.

In general, a unit consists of three pieces: an import specification, an export specification, and a body. The first two are just sequences of names. The body of a **unit** is, like the body of a program, a series of distinctly-named definitions followed by an expression. Imported variables are in scope in the body of the unit; they are resolved to values during unit linking. Each exported variable must be defined in the body of the unit. Imported variables cannot be directly exported, because this might lead to definitional cycles.
Here is a program fragment, extracted from a full-fledged Racket program,\(^2\) that uses all unit forms and operations:

```racket
(define world@ ((unit (import tock clack) (export key= ? big-bang))
               (define key= ? (λ (ke1) (λ (ke2) . . .))))
               (define big-bang (λ (w) . . . tock . . . clack . . .))
               "the default expression")

(define client@ ((unit (import key= ? big-bang) (export tock clack))
               (define tock (λ (w) . . .))
               (define clack (λ (ke) (λ (w) . . . (key= ? ke . . .) . . .)))
               (big-bang . . .))

(define cmain@ ((compound-unit (import) (export) world@ client@)))

:invoke-unit cmain@)
```

The unit named `world@`\(^3\) defines and exports two functions: `key= ?` (an equality predicate on keyboard events) and `big-bang` (which launches an interactive graphical program when applied to some value). In return, the `world@` unit imports from its clients two functions: `tock` (called in response to clock ticks) and `clack` (called in response to keyboard events).

The second definition introduces and names a unit that plays the role of a potential client to the `world@` unit. Dually to `world@`, this `client@` unit imports `key= ?` and `big-bang` and exports `tock` and `clack`. Also note that this unit’s last piece is an expression that invokes the imported `big-bang` function.

Next, the third definition illustrates the compounding of two units into a single module. The `compound-unit` expression allows the exports of `world@` to flow into `client@` and vice versa. Because the two satisfy each other’s export and import needs, the resulting compound unit does not import anything else. It could export values, if so desired, but this is independent of the rest of the program.

---

\(^2\)This program is a stripped down, unit-based version of Racket’s universe.ss teachpack, a library for teaching functional GUI programming to novices [Felleisen et al., 2009b].

\(^3\)The use of “@” is a Racket convention that helps readers identify the names of units.
Finally, the last expression invokes the import-less $cmain@$ unit. This action runs all the expressions of the given unit, which in this case are all the expressions of the two units that went into the construction of $cmain@$.

In Racket, compound units may link as many units as needed. Pragmatically this multi-unit linking mechanism is highly useful. From the perspective of a model, it adds nothing but notational complications. Also, Racket’s unit system supports an operation for invoking a unit and splicing its exported definitions into the local lexical scope. Handling this case, even with the later addition of contracts, is straightforward and does not add much to the exposition.

**Figure 3.3** Values and evaluation contexts for units

$$
\begin{align*}
pv ::= & \ n \mid b \mid s \\
nv ::= & \ pv \mid (\lambda (x) \ e) \\
nlv ::= & \ pv \mid u \\
v ::= & \ pv \mid u \mid (\lambda (x) \ e) \\
dv ::= & \ (define \ x \ v) \\
P ::= & \ (program \ dv \ ... \ D \ d ... \ e) \mid (program \ dv \ ... \ E) \\
D ::= & \ (define \ x \ E) \\
E ::= & \ [ ] \mid (E \ e) \mid (v \ E) \mid (op \ E \ e) \mid (op \ v \ E) \mid (begin \ E \ e) \\
       & \ (if \ E \ e \ e) \mid (invoke-unit \ E) \mid CU \\
CU ::= & \ (compound-unit \ (import \ x \ ...) \ (export \ x \ ...) \ E \ e) \\
       & \ (compound-unit \ (import \ x \ ...) \ (export \ x \ ...) \ v \ E)
\end{align*}
$$

**Figure 3.4** The semantics of the core language

$$
(\lambda (x) \ e) \ v \rightarrow e(x := v) \ [\text{Beta}] \\
(op v_1 \ v_2) \rightarrow \delta[op, v_1, v_2] \ [\text{Delta}] \\
(if \ #t \ e, e) \rightarrow e ; \ [\text{If-True}] \\
(if \ #f \ e, e) \rightarrow e ; \ [\text{If-False}] \\
(begin \ v \ e) \rightarrow e \ [\text{Begin}] \\
P[x] \rightarrow P[v] \ [\text{Var}] \\
\text{where } \ (\text{define} \ x \ v) \in P[x]
$$

3.1.3 Dynamic Semantics

A reduction semantics classifies the syntactic elements of a language as values and computations and then specifies via relations on the syntax how the latter reduce to the former. Here the specification of the relations involves reduction contexts, turning the relations into functions on the syntax. In sum, a reduction semantics specifies a machine whose states are complete programs and whose instructions are functions from programs to programs; the machine reduces complete programs to programs in canonical form, known as values.

Our semantics consists of two pieces: the usual reduction semantics for the core language and an extension that specifies the meaning of units and operations on units. Figure 3.3
and exports as the form must correspond to an export of either of the two unit values. Each exported variable of the imports and a list of exports. Each imported variable in a constituent unit must be listed together into a single unit. Like a unit form, the compound-unit form also has a list of imports and a list of exports. Each imported variable in a constituent unit must be listed in either the exports of the other constituent unit or the imports of the compound-unit form, but not both. This means that the exports of the compound-unit form and the exports of each unit value must be distinct. Each exported variable of the compound-unit form must correspond to an export of either of the two unit values.

The result of the compound-unit form is a new unit value that has the same imports and exports as the compound-unit form:

\[
\text{compound-unit (import } x_1 \ldots \text{) (export } x_2 \ldots \text{)}
\]

\[
\text{(define } x_1 e_1 x_2 \ldots e_2 \text{)}
\]

\[
\text{invoke-unit (import } x_1 \ldots \text{) (export } x_2 \ldots \text{)}
\]

\[
\text{(define } x_3 e_3 \text{)}
\]

\[
\text{program } d_1 \ldots d_n \text{ e}
\]

\[
\text{define } x_n \text{ }
\]

classifies a subset of our language’s syntax as values and defines evaluation contexts for programs, expressions, and unit expressions. Values are collected into four categories: \( pv \) for primitive values, \( nlv \) for non-\( \lambda \) values, \( nul \) for non-unit values, and \( v \) for all values.

Next, figure 3.4 introduces the reduction relations for the core language. The relations for the core model are straightforward. Relations that use the \( \rightarrow \) notation occur within program contexts \( P[\ ] \) but have no need to either examine their context or change it in any way. In contrast, the [Var] relation shows how free variables are resolved by finding the corresponding top-level definition in the program via a condition on the surrounding evaluation context.

The two reduction relations in figure 3.5 explain how units are compounded and invoked. The [Invoke] rule describes the process of invoking a unit value. The invocation process involves renaming the internal definitions and their references to fresh variables and lifting the resulting definitions to the program level. Then the body expression of the unit is evaluated in place of the invoke-unit form.

The [Compound] rule shows how compound-unit takes two unit values and links them together into a single unit. Like a unit form, the compound-unit form also has a list of imports and a list of exports. Each imported variable in a constituent unit must be listed in either the exports of the other constituent unit or the imports of the compound-unit form, but not both. This means that the imports of the compound-unit form and the exports of each unit value must be distinct. Each exported variable of the compound-unit form must correspond to an export of either of the two unit values.

The result of the compound-unit form is a new unit value that has the same imports and exports as the compound-unit form:
(compound-unit (import x y) (export a))
  (unit (import x y) (export a)
    (define a (λ (z) (+ x y)))
    "default 1")
  (unit (import x a) (export b)
    (define b (λ (w) (− x (a w))))
    "default 2")
→
  (unit (import x y) (export a)
    (define a (λ (z) (+ x y)))
    (define b (λ (w) (− x (a w))))
    (begin "default 1" "default 2")

This new unit value contains the definitions from both component units. It also contains the expressions from both units. These two expressions are sequenced using the begin form, corresponding to the order of the two units in the compound-unit form.

When the definitions from both units are combined, all the definitions and their uses from each unit are renamed. The following program illustrates why:

(program
  (define server1@
    (unit (import) (export n) (define n 3) 1))
  (define server2@
    (unit (import) (export n) (define n 4) 1))
  (define client@
    (unit (import n) (export) (* n 2)))
  (define comp1@
    (compound-unit (import) (export) server1@ client@))
  (define comp2@
    (compound-unit (import) (export) server2@ client@))
  (compound-unit (import) (export) comp1@ comp2@))

Both server1@ and server2@ provide different implementations of n, but neither comp1@ nor comp2@, which link these units to client@, export n. Then comp1@ and comp2@ are linked in the body of the program. If we did not rename the internal definitions of n and their uses in each unit at this step, then the resulting unit would contain two conflicting definitions of n.
In addition to renaming all the definitions, the reduction rule also renames the uses of each unit’s exports in the other unit and constructs the export definitions for the combined unit by defining each export to be the value of the appropriate renamed variable.

### 3.1.4 Unit Pragmatics

Units are particularly useful when a program needs the functionality of one and the same client module in the context of several different service modules or when a program must decide at run-time which module to link in.

Let us illustrate each case with examples from the code of the DrRacket program development environment [Findler et al., 2002]. DrRacket supports several text books with teaching languages. A teaching language restricts the syntax of some production language for the purposes of a particular part of a book. Furthermore teaching languages come in series—typically BSL (for beginning student language), ISL (intermediate), and ASL (advanced)—so that students gradually see more and more of the full language.

**Figure 3.6 Linking clients to many servers**

```scheme
(compound-unit (import) (export)
  ;; the parsers
  [PBSL (bsl-parser@)]
  [PISL (isl-parser@)]
  [PASL (asl-parser@)]
  ;; the evaluators
  [BSL (interpreter@ PBSL)]
  [ISL (interpreter@ PISL)]
  [ASL (interpreter@ PASL)]
  ;; the testing
  [TEST (unit (import) ([prefix "bsl:" eval]
                             [prefix "isl:" eval]
                             [prefix "asl:" eval]))
    (export)
    (define prog1 '((define i (λ (x) x)) (i 10)))
    ...
    ...(check
      10
      (bsl:eval prog1)
      (isl:eval prog1)
      (asl:eval prog1)) ...)

BSL ISL ASL])
```

Each teaching language is parsed to a common intermediate representation. The results flow into an interpreter (and other tools) with annotations so that, for example, run-time error messages use concepts known to students of the given language and nothing else. In this context, the common interpreter and each parser come in the form of a unit: *inter-

The last three export a `parse` function; the first one imports it and exports an `eval` function.

DrRacket’s test suite ensures that certain programs from, say, BSL, have the same outcome in all three levels. To achieve this, the test suite links `interpreter@` to all three parser units and runs all three `eval` functions on the same program: see figure 3.6. The sketch shows how `compound-unit` links three parsers to one interpreter to obtain three complete evaluators. Via renaming, all three evaluators are linked to a testing unit and become available simultaneously.

Naturally DrRacket allows programmers to switch from one teaching language to another without shutting down and restarting the IDE. That is, the system decides at run-time which units should be linked in. The IDE uses the currently selected language to decide which language front-end to link with the interpreter:

```
(define (get-interpreter current-selection)
  (compound-unit (import) (export run)
    (cond
      [(eq? current-selection 'beginner) bsl-parser@]
      [(eq? current-selection 'intermediate) isl-parser@]
      [(eq? current-selection 'advanced) asl-parser@]
      interpreter@))
)
```

Last but not least, DrRacket can also launch arbitrary front-ends by loading units at runtime:

```
(define (get-interpreter parser-path)
  (compound-unit (import) (export run)
    (load parser-path) interpreter@))
)
```

For further examples of how units provide useful software abstractions, see Findler and Flatt [1998], Flatt and Felleisen [1998], and Graunke’s work on the PLT web server [Graunke et al., 2001].

### 3.2 Contracts

While contracts for first-order functions in the context of static modules are easy to understand, contracts for higher-order functions and contracts for a dynamic module system demand a gradual introduction. In addition to a series of examples of units with contracts, this section develops desiderata for our contract system, including the ability to introduce contracts on a gradual basis.
3.2. CONTRACTS

3.2.1 Units with Contracts

Units are naturally the parties to contracts in our world (for notational simplicity, we do not introduce explicit interface definitions). We attach contracts directly to the import and export specifications of a unit, e.g.,

\[
\text{(define convert@)}
\]

\[
\text{(unit (import) (export \{convert \rightarrow \text{string} \rightarrow \text{number}? real?\})]
\]

\[
\text{(define convert \(\lambda (n) \ldots\))}
\]

\[
\text{"default"})}
\]

This first sample contracts states that \text{convert} maps strings convertible to numbers\(^4\) to reals. An alternate implementation of \text{convert@} could also export the predicate that protects \text{convert}:

\[
\text{(define convert@)}
\]

\[
\text{(unit (import) (export \{convertible? \rightarrow \text{string}? \text{boolean}?\}] [convertible? \rightarrow \text{convertible}? \text{real}?\})]
\]

\[
\text{(define convert \(\lambda (n) \ldots\))}
\]

\[
\text{(define convertible \(\lambda (x) \ldots\))}
\]

\[
\text{"default"})}
\]

To use the services of \text{convert@}, we must write a unit that imports the \text{convert} function:

\[
\text{(define cvrt-client@)}
\]

\[
\text{(unit (import \{convert \rightarrow \text{string} \rightarrow \text{number}? real?\}) (export) (convert 4))}
\]

Since both units use the same contract, we can obviously link them:

\[
\text{(compound-unit (import) (export) convert@ cvrt-client@)}
\]

Furthermore, it is also obvious that \text{convert@} is responsible for the positive positions of the contract, and \text{cvrt-client@} is responsible for the negative positions.

Units don’t have to use the same contract to be compatible, however. A programmer may just know that two units should be able to collaborate even if the contracts aren’t quite the same:

\[
\text{(define string-client@)}
\]

\[
\text{(unit (import \{convert \rightarrow \text{string}? \text{real}?\}) (export) (convert "hellow"))}
\]

Here \text{string-client@} imports a \text{convert} function that is expected to accept all strings. A programmer who believes that \text{string-client@} does not apply \text{convert} to inconvertible strings may still link the two units:

\[
\text{(compound-unit (import) (export) convert@ string-client@)}
\]

\(^4\)In Racket, \text{string\rightarrow number} translates some given string into a number, if possible, and produces false otherwise. Since all non-false values count as true, this function can act as a basic predicate, too.
Of course, an invocation of the result raises a contract error.

The question is which unit should be blamed. The \texttt{convert@} unit provides a \texttt{convert} function from number-convertible strings to numbers, and we may assume that it fulfills its contract. The \texttt{string-client@} unit applies \texttt{convert} to "hellow", which is appropriate for its contract. Clearly, neither \texttt{convert@} or \texttt{string-client@} should be blamed. This leaves us with the compound unit, which becomes a third party to the contracts, an implicit adapter.

### 3.2.2 Adding Contracts Gradually

Programmers use dynamic languages because they wish to get something running quickly with as little overhead as possible. Hence language designers must be prepared to deal with programs that mix modules with and without contracts.

Let us revisit the preceding example with the contracts from \texttt{string-client@} removed:

\begin{verbatim}
(define plain@ (unit (import convert) (export) (convert -4)))

(define c@ (compound-unit (import) (export) convert@ plain@))
\end{verbatim}

If the program then invokes \texttt{c@}, a contract error is raised due to the contract on \texttt{convert@}, which was not removed.

As mentioned, we assume that \texttt{convert@} lives up to its contract. The \texttt{plain@} unit does not subscribe to a contract for \texttt{convert} (or anything else) and thus can’t be the target of a “blame message.” Again we are forced to blame the linking \texttt{compound-unit} expression, because it allowed the \texttt{convert} function, which was exported with a contract, to flow to a party that misapplies the function.

In short, we are forced to conclude:

- All units guarantee the negative positions of their import contracts and the positive positions of their export contracts.

- Compound units guarantee for their constituent units the positive positions of the import contracts and the negative positions of the export contracts.

### 3.2.3 A Contract Combinator for Units

Because units are first-class values, our language also needs a method of ascribing contracts to units in module interfaces. Consider the following unit:
(define combine-convert@
  (unit (import) (export [combine ...])))

(define combine
  (λ (u@)
      (invoke-unit
       (compound-unit (import) (export) u@ convert@))))

It exports a function that consumes a unit, links it with the above convert@ unit, and invokes the result.

To protect the combine export of combine-convert@, we need a contract combinator that guards units. This contract combinator must be able to describe those details important for interfacing with a unit, i.e., the imports and exports. In Racket, unit/c plays this role. For the given example, the combinator would be used as follows:

(export [combine (→ (unit/c (import [convert (→ string→number real?)]) (export)) real?) ...]

A unit that is being guarded with a unit contract must import a subset of the imports listed by the unit/c expression and export a superset of the exports. Furthermore, we must decide whether the unit which results from applying a unit contract has the same imports and exports as the original unit, or whether it has the same imports and exports as those listed in the contract. Here we take the latter view, as contracts constitute a promise of how a value is utilized, and thus assuming unlisted imports or exports is inappropriate.

3.3 Units with Contracts

Equipped with an informal understanding of contracts in a unit setting, we proceed to formulate a model of this world. First, we extend the syntax of our model to accommodate contracts. Second, we formulate a model of contract monitoring. Third, we state two essential theorems about the model: one concerning “type soundness” and one for “contract soundness.”

3.3.1 Syntax for Units with Contracts

Figure 3.7 spells out our revisions to the syntax of section 3.1 that add contracts to our model. The first clause specifies contracts as unit contracts, functions contracts, or (expressions that evaluate to) arbitrary predicates. The second line adds contracts to the import and export specifications of units. The second and third line add names to both the unit and compound-unit forms to represent blame; a production system would naturally use
source locations instead. Figure 3.8 presents our running example from section 3.1—a GUI module and its clients—in this revised syntax.

**Figure 3.7** Surface syntax extensions for contracts
\[ c ::= (\text{unit}\ c\ (\text{import}\ [x\ c\ ...]\ (\text{export}\ [x\ c\ ...]\)\ !\ (\rightarrow\ c)\ !\ e) \\
    u ::= (\text{unit}\ s\ (\text{import}\ [x\ c\ ...]\ (\text{export}\ [x\ c\ ...]\ d \ ...\ e) \\
    cu ::= (\text{compound-unit}\ s\ (\text{import}\ [x\ c\ ...]\ (\text{export}\ [x\ c\ ...]\) e\ e) \]

**Figure 3.8** Extended example with predicate exports
\[
\text{(program)} \\
\quad (\text{define any/c}\ (\lambda\ (x)\ \#t)) \\
\quad (\text{define world@}\\n\qquad (\text{unit}\ "world") \\
\qquad \quad (\text{import}\ [\text{world}?\ (\rightarrow\ any/c\ any/c)] \\
\qquad \quad \quad [\text{tock}\ (\rightarrow\ \text{world}?\ \text{world}?)] \\
\qquad \quad \quad [\text{clack}\ (\rightarrow\ \text{key}?\ (\rightarrow\ \text{world}?\ \text{world}?)]) \\
\qquad (\text{export}\ [\text{key}?\ (\rightarrow\ any/c\ any/c)] \\
\qquad \quad [\text{key}=?\ (\rightarrow\ \text{key}?\ (\rightarrow\ \text{key}?\ any/c)]] \\
\qquad \quad [\text{big-bang}\ (\rightarrow\ \text{world}?\ any/c)]) \\
\quad \quad (\text{define key}\ (?\ (\lambda\ (ke)\ ...) )) \\
\quad \quad (\text{define key}=?\ (?\ (\lambda\ (ke1)\ (\lambda\ (ke2)\ ...)))) \\
\quad \quad (\text{define big-bang}\ (\lambda\ (w)\ ...\ \text{tock}...\ \text{clack}...\) 1)) \\
\quad (\text{define client@}\\n\qquad (\text{unit}\ "client") \\
\qquad \quad (\text{import}\ [\text{key}?\ (\rightarrow\ any/c\ any/c)] \\
\qquad \quad \quad [\text{key}=?\ (\rightarrow\ \text{key}?\ (\rightarrow\ \text{key}?\ any/c)]] \\
\qquad \quad [\text{big-bang}\ (\rightarrow\ \text{world}?\ any/c)]) \\
\qquad (\text{export}\ [\text{world}?\ (\rightarrow\ any/c\ any/c)] \\
\qquad \quad [\text{tock}\ (\rightarrow\ \text{world}?\ \text{world}?)] \\
\qquad \quad [\text{clack}\ (\rightarrow\ \text{key}?\ (\rightarrow\ \text{world}?\ \text{world}?)])] \\
\quad \quad (\text{define world}\ (?\ (\lambda\ (w)\ ...))) \\
\quad \quad (\text{define tock}\ (?\ (\lambda\ (w)\ ...))) \\
\quad \quad (\text{define clack}\ (?\ (\lambda\ (ke)\ (\lambda\ (w)\ ...\ \text{key}=?\ ke\ ...\)))) \\
\qquad (\text{(big-bang} 0))) \\
\quad (\text{invoke-unit} \\
\qquad (\text{compound-unit} \ "\text{linker}\)\ (\text{import}\ (\text{export}\ \\
\qquad \quad \quad \text{world@}\ \text{client@})
\n\textbf{3.3.2 The Idea}

Findler and Felleisen [2002] describe a model of higher-order contracts and static modules. Their model assumes that programs consist of a sequence of modules and a main expression. Contract checking in this model compiles each reference to an imported function into a guard expression. Roughly speaking guard expressions wrap imported functions with a contract and information about the parties to the contract. The wrapper knows how to check every kind of value, including higher-order functions.
3.3. **UNITS WITH CONTRACTS**

In a unit world, however, the compiler lacks two critical pieces of information to use this strategy. First, while units specify their imports, they can’t possibly know the contracts that the exporting service module imposes on them. Second, client units don’t know which server unit provides the services. Hence, the compiler doesn’t know the name for the positive blame positions in a function contract.

To overcome these knowledge gaps, our compiler for unit contracts uses a different compilation strategy for exports and delays additional aspects of contract checking to the linking step. Specifically, it adds guards to the exported functions as well as imported ones. Furthermore, it uses a protocol for introducing and changing placeholders for the positive blame positions.

![Figure 3.9](image)

**Figure 3.9** Internal syntax for guard expressions

\[ e ::= \ldots \| (\text{guard } e \ c \ e) \| (\text{error } s) \]

Figure 3.9 shows the extensions to the expression syntax with guard expressions and contract errors. In our model, these new syntactic forms are invisible to the programmer; an implementation can re-use existing syntax (e.g., \texttt{if} expressions and \texttt{error} functions). The \texttt{guard} form consists of an expression, a contract, and two strings (blame labels) for the positive and negative blame positions of contracts, respectively.

### 3.3.3 Contract Compilation

Figure 3.10 specifies the compiler of our unit contract model. The metafunction \( \xi \) is applied to all top-level expressions in the program. It is mostly a straightforward homomorphism on expressions and contracts except for the \texttt{unit} case. For units, \( \xi \) renames all internal definitions and their uses, creates (using \( \rho c \)) new export definitions that guard the internal implementation, and replaces all uses of import variables with guarded versions of the same. Finally, the function adds a definition for \( x_s \) to unit bodies; this new definition acts as a placeholder for the blame label of the yet-to-be-determined compound unit that links this unit into the full program.

The trickiest part of this conversion is handling the use of imported identifiers in contracts. Since uses of variables in contracts are considered to be internal to the unit, we do not check the contracts for exported variables, but we do need to check those on imports. Contracts on imports may use other imports, however, and to deal with such uses, our compiler unrolls the contracts for imports and exports completely.

The metafunction \( \Sigma \) appropriately unrolls these contracts. As defined, this unrolling may not terminate if two or more imports use each other in their respective contracts:

\[
(\text{unit } "\text{loop}" (\text{import } [x? (\rightarrow y? \ any/c)] [y? (\rightarrow x? \ any/c)]) (\text{export}) (x? 3))
\]
Figure 3.10 Contract compilation

\[ \xi[\text{import } x, c, \ldots] = (\text{import } x, c, \ldots) \]
\[ \xi[\text{export } x, c, \ldots] = (\text{export } x, c, \ldots) \]
\[ \xi[\text{contract } u] = (\text{contract } u) \]
\[ \xi[\text{where } u = \text{unit } \ldots] = (\text{where } u = \text{unit } \ldots) \]
\[ \xi[\text{if } e, \ldots] = (\text{if } e, \ldots) \]
\[ \xi[\text{begin } e, \ldots] = (\text{begin } e, \ldots) \]
\[ \xi[\text{unit } \ldots] = (\text{unit } \ldots) \]
\[ \xi[\text{compose } \ldots] = (\text{compose } \ldots) \]
\[ \xi[\text{contract } \text{import } x, c, \ldots] = (\text{contract } \text{import } x, c, \ldots) \]
\[ \xi[\text{contract } \text{export } x, c, \ldots] = (\text{contract } \text{export } x, c, \ldots) \]
\[ \xi[\text{contract } \text{where } u = \text{unit } \ldots] = (\text{contract } \text{where } u = \text{unit } \ldots) \]
\[ \xi[\text{contract } \text{if } e, \ldots] = (\text{contract } \text{if } e, \ldots) \]
\[ \xi[\text{contract } \text{begin } e, \ldots] = (\text{contract } \text{begin } e, \ldots) \]
\[ \xi[\text{contract } \text{unit } \ldots] = (\text{contract } \text{unit } \ldots) \]
\[ \xi[\text{contract } \text{compose } \ldots] = (\text{contract } \text{compose } \ldots) \]

where \( x \), \( y \), \( z \), \( a \), \( b \), \( c \), \( d \), \( e \), \( f \), \( g \), \( h \), \( i \), \( j \), \( k \), \( l \), \( m \), \( n \), \( o \), \( p \), \( q \), \( r \), \( s \), \( t \), \( u \), \( v \), \( w \), \( x \), \( y \), \( z \) are fresh identifiers.
3.3. UNITS WITH CONTRACTS

An implementation of $\Sigma$ can detect such cyclic dependencies in import contracts for each
**unit** or **compound-unit** expression and refuse to compile the program if such a cycle is
detected. We skip this complication for $\Sigma$ because it adds little to this presentation.

### 3.3.4 Dynamic Contract Checking

The rest of the contract checking process is performed at run-time for both higher-order
functions and first-class units. The former follows Findler and Felleisen [2002]. The latter
demands changes to the reduction rules for units.

Figure 3.11 specifies the revised reduction relations, along with the revised definitions of
evaluation contexts. These new relations rely on a number of auxiliary functions, including
substitution and renaming functions ($\psi$). The latter are defined in figure 3.12.

Two aspects of this process deserve special attention: unit linking and unit guarding.
The rest of this section is dedicated to these two aspects of run-time checking.

#### 3.3.4.1 Linking Units

As noted in section 3.3.2, compounding units introduces new parties to contracts, especially
for contracts that don’t match. To assign blame to the proper compound unit, our run-
time system must synthesize guard expressions during a linking step. A revised [Compound]
reduction relation achieves this as follows:

- create a new placeholder blame label for future linking;
- replace the placeholder blame label from each unit with the blame label of the com-
  pound unit;
- rename the definitions from each unit and their uses, which includes imported uses in
  the other unit;
- guard all uses of the compound unit’s imports;
- create guarded versions of the compound unit’s exports; and
- sequence the body expressions.

The renaming step ensures that linking does not inadvertently add contract checks in a
place where they are unwanted.

Consider this program:
Figure 3.11 New and revised reduction rules

\[ E ::= \ldots \mid (\text{guard } E \vee e) \mid (\text{guard } v \in E) \mid (\text{guard } v \in E) \]

\[ CU ::= (\text{compound-unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \ E) \]
\[ \mid (\text{compound-unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \ v \ E) \]

\[ P[(\text{invoke-unit } s)] \]
\[ \rightarrow \text{add-defs}[P, \{(x_0 \ldots) := (x_0 \ldots)\}] \mid ((\text{define } x, e_0[(x_0 \ldots) := (x_0 \ldots)]) \}) \]]

where \( u = (\text{unit } s \ (\text{import } [x\ c] \ldots) \ (\text{define } x, e_0 \ldots) \ e_0), \)
\( (x_0 \ldots) \text{ fresh} \)

\[ \text{(compound-unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \]
\[ \text{(define } x, \ldots \text{ "unknown")} \ (\text{define } x, e_0 \ldots) \ e_0) \]

\[ \rightarrow (\text{unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \]
\[ \text{(define } x, \ldots \text{ "unknown")} \ (\text{define } x, e_0 \ldots) \ e_0) \]

\[ d_1 \ldots d_i \ldots \text{pc} \left[ x, c, (x_i_0 x_i_1 \ldots x_i_k x_i \ldots), x, x_i \ldots \right] \]
\[ \text{(begin } e, e_0)) \]

where \( (c_0 \ldots) = (\text{Sigma}[c, (x_0 \ldots)], s, s\{(x_0 \ldots) := (x_0 \ldots)\}), \)
\( (c_0 \ldots) = (\text{Sigma}[c, (x_0 \ldots)], s, s\{(x_0 \ldots) := (x_0 \ldots)\}), \)
\( (e_0 \ldots) = (\text{Sigma}[e, (x_0 \ldots)], s, s\{(x_0 \ldots) := (x_0 \ldots)\}), \)
\( (e_0 \ldots) = (\text{Sigma}[e, (x_0 \ldots)], s, s\{(x_0 \ldots) := (x_0 \ldots)\}), \)
\( (d_1 \ldots) = \text{Psi}[\text{define } x, e_0 \ldots], (x, x_i), (x, x_i \ldots)], \)
\( (d_2 \ldots) = \text{Psi}[\text{define } x, e_0 \ldots], (x, x_i), (x, x_i \ldots)], \)
\( (x, x_i, x_i, x_i \ldots) \text{ distinct}, \; (x_0 \ldots) \subseteq (x, x_i) \ldots, \)
\( \{x_2 \ldots \subseteq (x, x_i) \ldots; \{x, x_i \subseteq (x, x_i) \ldots, \}
\)
\( x, x_i, x_i, x_i \ldots) \text{ fresh} \)

\[ (\text{guard } (1_{(x) e} \rightarrow c, c), s, s_0) \]

\[ \rightarrow (1_{(x) e} \text{ (guard } (k_{(x) e} \text{ (guard } s \in c; s_0)) \ (c, c, s_0))) \]

\[ (\text{guard } n_{lv} \rightarrow (c, c), s, s_0) \rightarrow \text{error } s_0) \]

\[ \text{(unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \]
\[ \text{(define } x, e_0 \ldots) \ e_0) \]

\[ \rightarrow (\text{unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \ s, s_0) \]

\[ \text{(begin } e_0 \ldots) = (\text{Sigma}[e_0 \ldots], s, s_0 \ldots) \}
\[ (x, x_i, x_i, x_i) \text{ fresh} \]

\[ \text{(guard } (\text{unit } s \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \ d \ldots e_0) \]
\[ \text{(unit } c \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \ s, s_0) \]

\[ \rightarrow \text{(error } s_0) \]

where otherwise

\[ (\text{guard } n_{lv} \text{ (unit } c \ (\text{import } [x\ c] \ldots) \ (\text{export } [x\ c] \ldots) \ s, s_0) \]

\[ \rightarrow \text{(error } s_0) \]

\[ (\text{guard } v \ e, s, s_0) \rightarrow (\text{if } (v \ e) \ (\text{error } s_0)) \]

\[ P[(\text{error } s)] \rightarrow \text{(error } s) \]

\[ \text{[Error]} \]
Here, for example, we do not want the use of $f$ in \texttt{client@} to be checked with the export contract from \texttt{link@}.

The compiler translates the program as follows:

\[
\text{program (define ge/c (λ (m) (λ (n) (< = m n)))})
\]

\[
(\text{define server@})
\]

\[
(\text{define "server" (import) (export [f (→ (ge/c 2) (ge/c 4))])})
\]

\[
(\text{define f (λ (n) (* n n))) "default")
\]

\[
(\text{define client@})
\]

\[
(\text{define "client" (import [f (→ (ge/c 2) (ge/c 4))]) (export) (f 2))}
\]

\[
(\text{define link@})
\]

\[
(\text{define "link" (import) (export [f (→ (ge/c 3) (ge/c 9))])}
\]

\[
\text{server@ client@}) \ldots)
\]
(program (define ge/c (λ (m) (λ (n) (<= m n)))))

(define server@
  (unit "server" (import) (export (f → (ge/c 2) (ge/c 4)))))
  (define x "unknown")
  (define fl (λ (n) (* n n)))
  (define f (guard fl (→ (ge/c 2) (ge/c 4)) "server" x))
  "default")
)

(define client@
  (unit "client" (import (f → (ge/c 2) (ge/c 4)))) (export)
    (define x "unknown")
    (guard f (→ (ge/c 2) (ge/c 4)) x "client") 2))
)

(define link@
  (compound-unit "link" (import) (export (f → (ge/c 3) (ge/c 9)))))
  server@ client@)
)

After substituting the two units into the compound-unit expression, the [Compound] relation applies and linking takes place:

(program (define ge/c (λ (m) (λ (n) (<= m n)))))

(define server@
  (unit "server" (import) (export (f → (ge/c 2) (ge/c 4)))))
  (define x "unknown")
  (define f1 (λ (n) (* n n)))
  (define f (guard f1 (→ (ge/c 2) (ge/c 4)) "server" x))
  "default")
)

(define client@
  (unit "client" (import (f → (ge/c 2) (ge/c 4)))) (export)
    (define x "unknown")
    (guard f (→ (ge/c 2) (ge/c 4)) x "client") 2))
)

(define link@
  (unit "link" (import) (export (f → (ge/c 3) (ge/c 9)))))
    (define x "unknown")
    (define x1 "link")  (define x2 "link")
    (define f2 (λ (n) (* n n)))
    (define fl (guard f2 (→ (ge/c 2) (ge/c 4)) "server" x1))
    (define f (guard f1 (→ (ge/c 3) (ge/c 9)) "link" x))
    "default"
    ((guard f1 (→ (ge/c 2) (ge/c 4)) x2 "client") 2)))) . . . )
The use of $f$ from the body of $\text{client}@$ has been appropriately altered to refer to the definition $f1$, which corresponds to the guarded export of $\text{server}@$, and not $f$, which is the guarded export of $\text{link}@$.

### 3.3.4.2 Guarding Values with unit/c

A unit/c contract requires a few first-order run-time checks. Specifically, the [UC-Unit] reduction ensures that the given value is a unit and that it has the required exports and imports. For other cases, the remaining UC rules pinpoint and explain the violation.

The result of a [UC-Unit] reduction is a unit that contains the additional contract checks on its imports and exports. During its construction, the reduction renames the definitions inside the unit. It also replaces all uses of imports with appropriate guard expressions. Finally, it creates new definitions to guard the exports listed by the unit/c contract. The new unit is limited to those exports and to imports listed in the unit/c expression. Those imports that are listed in the latter, but not imported by the original unit are added to the import list with the contract that always succeeds.

### 3.4 Implementation

While a model is a mathematically precise explanation of a language extension, it often fails to bring across how to add such extensions to an existing implementation. For example, Flatt and Felleisen [1998] explain units with a reduction semantics that copies the bodies of units at will, which would impose a huge overhead if translated naively into a compilation strategy. Instead, the unit compiler represents all units as grey boxes and implements unit linking and invocation via operations on these grey boxes.

In this section, we sketch how to implement our contract model in the context of a unit system such as the one available in Racket. Other dynamic module systems may need a different treatment, but we expect that the sketch here provides some insight on how to adapt the model to other contexts. The section starts with a brief, cursory explanation of how units are implemented, especially how linking is implemented without copying code. In the second subsection, we explain how to add contracts to such a unit implementation. Finally, in the last section, we discuss how to implement unit/c as a projection [Findler and Blume, 2006].
3.4.1 Implementing Units

At a high-level of abstraction, the Racket compiles all units into thunks—parameter-less procedures—that produce two values:

- a mapping \( M \) from exports to reference cells; and
- a function \( f \) that consumes a mapping from imports to reference cells and runs the unit’s body.

The compiler replaces uses of the imports in the unit body with appropriate accesses into the import mapping. It also adds assignment statements below each definition of an export that transfers the value into the expected reference cell of \( M \).

Given a thunk-based representation of units, a unit invocation proceeds as follows. An application of the thunk yields two values, including a function that consumes an import mapping and runs the body of the unit. Since units are only invoked if their import list is empty, this function is applied to the empty import mapping and then executes the definitions and expressions of the given unit.

Similarly, the purpose of linking two (or more) units is to construct a new thunk from the given thunks. This new thunk returns \( M \) and \( f \), which are constructed in a three-step process:

1. It applies the unit thunks for the constituent units, resulting in two export mappings \((M_1, M_2)\) and two body functions \((f_1, f_2)\).
2. Next a new export mapping \( M \) is constructed from the contents of \( M_1 \) and \( M_2 \).
3. In addition, a new body function \( f \) is constructed. It
   
   a) receives the import mapping \( I \) for the compound unit,
   b) constructs an import mapping \( I_1 \) from \( I \) and \( M_2 \),
   c) applies \( I_1 \) to \( f_1 \),
   d) constructs an import mapping \( I_2 \) from \( I \) and \( M_1 \), and
   e) applies \( I_2 \) to \( f_2 \).

Note that the process deals with the bodies of the constituent units as black boxes, copying not the code in these units but only pointers to the code.
3.4. IMPLEMENTATION

3.4.2 Implementing Unit Contracts

Although our model of unit contracts assumes that linking may access the unit bodies, doing so for an implementation would radically alter the way Racket deals with units. Since we do not wish to fall back on a copying compiler, we add contracts only during linking, not to atomic units. Adding the appropriate contract guards in this manner poses challenges, however, because the values being guarded are set in the reference cells only after the unit bodies have been executed.

Our compiler therefore changes how export and import mappings work. Specifically, the body function for a unit fixes the import mappings such that when the imports are accessed, a value is received that has been appropriately wrapped with contracts. This wrapping must be delayed at least until the first time the import is accessed, to ensure it is not prematurely requested.

To allow for this delayed wrapping of imports, we alter the translation of units so that exports are mappings from names to thunks. The export thunks of atomic units are closed over a reference cell and simply return the value currently stored. Imports in a unit body are just translated to an access into the import mapping and application of the resulting thunk.

When the function $f$ corresponding to a compound unit constructs the import mappings $I_1$ and $I_2$, it creates new thunks for names coming from the export mapping of the other constituent unit ($M_2$ and $M_1$, respectively). This thunk evaluates the original thunk from that export mapping, resulting in the exported value, and then wraps that value with the appropriate contracts. The contracts are those listed by both the exporting and importing units.

3.4.3 Implementing unit/c

Following Findler and Blume [2006], unit/c denotes a pair of projections. For function contracts, a projection consumes positive and negative blame labels and returns a function, which is like the given one, except that it uses the projections to enforce contracts.

The projection for a unit/c contract returns a function that

1. checks that its input is a unit;

2. checks that the unit imports a subset and exports a superset of those names listed in the unit/c form;

3. applies the unit thunk to get $M$ and $f$;
4. and constructs a unit thunk that constructs a new export mapping \( M_n \) and function \( f_n \) and returns both.

The new export mapping \( M_n \) contains entries for those names listed in the \texttt{unit/c} form and maps them to new thunks. Each new thunk applies the thunk from \( M \) to get the exported value, applies the positive and negative blame labels to the projection corresponding to that name’s contract, and then returns the application of the projection result to the exported value.

The new function \( f_n \) takes its import mapping argument \( I \) and performs almost the same alteration as for \( M_n \) to create \( I_n \). The only difference is that the positive and negative blame are swapped when applied to the projections. When this step is finished, \( f_n \) then applies \( f \) to \( I_n \).

### 3.5 Contracts for Nominally Linked Units

Racket’s original implementation of units were structurally linked as described in the previous systems. However, in the current implementation [Owens and Flatt, 2006] units are nominally linked via signatures, which name collections of variables for import or export from a unit. The first part of this section illustrates signatures and describes the addition of contracts to unit signatures. The second and third subsections then present examples of uncontracted and contracted units. The last subsection explains how to implement units as contract boundaries in Racket and how the addition of contracts affects our implementation.

#### Figure 3.13 Signatures with contracts

```
#lang scheme
(define-signature world
  (key
    key=?
    big-bang))

(define-signature client
  (world?
    tock
    clack))

...  

#lang scheme
(define-signature world/c
  ((contracted
    [key? (→ any/c boolean?)]
    [key=? (→ key? key? boolean?)]
    [big-bang (→ any/c void?)])

(define-signature client/c
  ((contracted
    [world? (→ any/c boolean?)]
    [tock (→ world? world?)]
    [clack (→ world? any/c world?)])

...  
```
3.5. CONTRACTS FOR NOMINALLY LINKED UNITS

3.5.1 Signatures and Contracts

A unit *signature* is a second-class named collection of variables. Units use sequences of signatures to specify their imports and exports. An exported signature can satisfy an import requirement for another unit only if that unit imports the signature with the *same* name. In other words, the unit system uses nominal matching.

For our examples, we use the two signatures on the left side of figure 3.13. These signatures describe interfaces that are useful for implementing interactive animations in a world-passing style [Felleisen et al., 2009b]. The *world* signature contains three names: *key?*, which is a predicate that determines whether a value is a keyboard event; *key=?!*, which is an equivalence predicate; and *big-bang*, which launches an animation when applied to a world (*world?*). The *client* signature also contains three names: *world?*, which is a predicate on worlds; *tack*, which is an event handler for clock ticks, mapping worlds to worlds; and *clack*, which is an event handler for keyboard events, from worlds and keyboard events to worlds.

Naturally, programmers wish to express such specifications as contracts in order to protect units. We have therefore extended the language of signatures with the *contracted* keyword, which combines signature variables with contracts. The right hand side of figure 3.13 shows the contracted versions of the signatures. Notice that signature contracts can involve elements of the same signature.

3.5.2 Units without Contracts

The import signatures of a unit introduce bindings for all their variables for the unit body; conversely, if a unit exports a signature, it must define all the variables listed in the signature. Figure 3.14 contains some sample units that utilize the uncontracted signatures from the preceding subsection.

When *compound-unit* is used to link a collection of units, the exported definitions from one unit are typically used to satisfy import requirements for one or more of the other units. Thus we can link *client@* and *world@* like this:

```
(define pgrm@
   (compound-unit/infer (import) (export) (link world@ client@)))
```

The “infer” suffix is a variant of *compound-unit* that infers how to wire up the exports and imports of the constituents.

---

5The * character at the end of signature names is merely a convention.
6As with *, the use of @ is a naming convention for units.
In general, the result of linking is a unit that has its own list of imports and exports and whose body is a sequence of the constituent unit bodies in the order listed in the link clause. The exports of the compound unit are satisfied from the exports of the constituent units, and the imports of the compound unit may be used to satisfy imports of the constituents. In contrast to modules, units can thus be compounded hierarchically, and they may refer to each other’s exports and imports in a mutually referential manner.

Finally, units with empty import signatures can be invoked, e.g.

```
(invoke-unit pgrm@)
```

The effect is to execute the body of world@, which consists entirely of definitions, and then to execute the body of client@, which calls big-bang.

### 3.5.3 Units with Contracts

The use of signatures with contracts turns units into blame parties and their boundaries into contract boundaries. In the following code, the definitions of world@ and client@ differ from the earlier definitions only in the import/export specification, and so we elide the bodies:

```
(define-unit world@ (import client/`c`) (export world/`c`) ...)
(define-unit client@ (import world/`c`) (export client/`c`) ...)
```

When we link client@ and world@ and invoke the result:
(invoke-unit
   (compound-unit/infer (import) (export) (link world@ client@)))

then client@ is blamed if either tock or clack cause the world to become negative or increase beyond 500.

The signatures world/c^ and client/c^ illustrate that a contract in a signature may refer to other elements from the same signature. Thus, we must decide how these contracts interact with the linked units’ contract boundaries. In particular, we must decide whether references to signature elements within contracts are guarded or not. For the purposes of this work, we consider all signature contracts as occurring within the importing unit’s contract boundary and therefore the compiler guards all uses of contracted signature elements inside those contracts. This ensures that exported variables are not misused by the contracts and concurs with our implementation strategy.\footnote{This design decision is overly conservative and deserves to be revisited once we have enough experience with our new contract system. Furthermore the current contract system does not permit programmers to use elements from one signature in a different signature for the specification of contracts. Extending the contract system in this direction may also force us to revisit the design decision on how to check contracted functions within contracts.}

### 3.5.4 Implementing Units as Contract Boundaries

Adding contracts to the unit system poses several challenges. First, units do not enter a contract with a known party; instead they specify via signature contracts what they expect from their context. Second, the same unit may be linked to several different units at runtime and may thus enter contracts with several different parties. Hence, the compiler cannot pass on enough knowledge about the contract parties to the run-time checks. Third, due to nominal linking, a compound unit may only link constituent units whose contracts are identical. Therefore blame labels can be exchanged as units are linked.

The first part explains the addition of signature-based contracts to the existing implementation. The second part covers additional features of the unit system.

#### 3.5.4.1 Contracts in Signatures

Since units must agree on their shared signatures \textit{by name} and since we add contracts to signatures, linked units automatically agree on all of the contracts of the shared variables. That is, a compound unit cannot possibly link two units whose contracts don’t match, as in figure 3.15. Thus, it is impossible for the linker to assume any responsibility for contract errors. Put differently, there is no need for checking contracts within the compound unit and it need never be blamed. Put positively, our implementation limits blame to the exporting unit and the importing unit.
CHAPTER 3. CONTRACTS FOR FIRST-CLASS MODULES

Figure 3.15 Mismatched signatures and contracts

```
#lang scheme
(define-signature lexer* ((contracted [lex -> string? (listof token?)])))
(define-signature lexer2* ((contracted [lex -> input-port? (listof token?)])))
(define-signature parser* ((contracted [parse -> string? ast?])))
(define-unit lexer@ (import) (export lexer2*)
  (define (lex str) . . . ))
(define-unit parser@ (import lexer*) (export parser*)
  (define (parser str) (let ((tokens (lex str)) . . .))
  (compound-unit/infer (link lexer@ parser@))]
```

The key to our addition of contracts is to separate the translation of contracted signature variables from those of uncontracted ones. For contracted exports, the compiler generates code that sets the cell for the exported value to a structure with two fields:

- one for the value of the exported variable, and
- one that uniquely identifies the exporting unit, i.e., its blame label.

When the compiler encounters a contracted import, it deconstructs this kind of structure and retrieves the contract from the imported signature. From these two pieces, the compiler constructs an appropriate guard expression for the imported value. This contract-guard uses the export blame label for positive blame report and the importing blame label for negative blame reports.8

3.5.4.2 Structural Linking

The unit system supports another important linguistic construct. One form, unit/s, provides a mechanism for linking units structurally. This provides backwards compatibility for use with the earlier implementation of units in Racket [Flatt and Felleisen, 1998].

The unit/s form takes import and export specifications as well as a unit value and creates a new unit value. Its imports and exports must structurally match the imports and exports of the given unit value; the resulting unit value uses the given imports and exports and the given unit’s body. Since this operation on units changes the import and export signatures, the contracts on the imported and exported values may be inappropriate for the original unit. Hence, the compiler must introduce contract checks into the result of unit/s that blames the new unit value when contract mismatches occur, instead of allowing either the original unit value or any unit with which it is linked to be blamed. The introduction of these contract checks is similar to the implementation of unit/c described in section 3.4.

8Roughly speaking, it applies two projections to the value: one for its “elimination” (negative) and one for its “introduction” (positive). If something goes wrong with the negative position, the client is blamed; otherwise the server is blamed. For details on the general idea, see Findler and Blume [2006].
Chapter 4

Contracts for First-Class Classes

In this chapter, we describe the Racket class system, which allows classes to be used as values and thus supports advanced methods for abstracting over the class hierarchy. We then design a contract system that protects individual classes, keeping in mind all the important features of the class system. With the contract system in hand, we evaluate the design in two ways: we model the class and contract system in order to prove guarantees about the contract system’s behavior, and we implement the contract system in Racket to show that it is performant.

In addition, we present the design of a contract system that protects sets of classes via common interfaces. First, we describe interfaces in the Racket class system. We then add contracts to interfaces, paying special attention to how blame is apportioned between classes, interfaces, and class clients. Finally, we describe our implementation and show the overhead induced by our changes to the class system.

4.1 Designing Class Contracts

Racket includes a class system [Flatt et al., 2006] loosely related to those found in languages like Java or C#. In Racket, however, classes are first-class values, enabling the programmer to abstract over classes and patterns in the class hierarchy.

We begin our survey of the Racket class system with this sample class definition:
(define fish
  (class object\%
    (super-new)
    (init-field name)
    (field [weight 10])
    (define max-weight 20)
    (define/public (full?)
      (\(\geq weight max-weight\)))
    (define/public (eat f)
      (set! weight (\(+ weight (get-field weight f)\))))))

The first expression in a class form specifies the superclass. In Racket, object\% evaluates to the root of the class hierarchy. The call to super-new initializes the superclass. The init-field clause lists expected initialization arguments, here a name for the fish, and also creates public fields for the provided values. The field clause contains the names of other public fields and their initialization expressions. Definitions using the keyword define describe private fields. The keyword define/public designates a public method. Of particular note, fields or methods defined by the class may be accessed within the class body as if they were local value or function definitions.

Here is a subclass of fish:

(define noisy-fish
  (class fish
    (super-new)
    (define/override (eat f)
      (printf "Eating a\n" (get-field name f))
      (super eat f)))))

It overrides the definition of eat to print a description of the given food. Like in C\#, a Racket class must explicitly override a public method. In Racket, the superclass may not be statically determinable; the definition helps the class system check at runtime that the superclass contains that public method, when noisy-fish is evaluated. Likewise, super calls may designate only methods that the current class overrides.

The superclass of the class form allows arbitrary expressions. Thus, we represent a mixin as a function that creates a new subclass of its argument:
4.1. DESIGNING CLASS CONTRACTS

```
(define (make-greedy fc)
  (class fc
    (inherit eat full?)
    (super-new)
    (define/public (eat-until-full lof)
      (cond [(full?) (printf "Burp!\n")]
           [(null? lof) (void)]
           [else (eat (first lof))
             (eat-until-full (rest lof))]))
  )
)

(define greedy-fish (make-greedy fish))
(define greedy-noisy-fish (make-greedy noisy-fish))
```

The second definition applies the `make-greedy` mixin from the first definition to the `fish` class, and the third one uses it on `noisy-fish`. The `inherit` clauses in the mixin require the superclass to have the listed public methods and allow the subclass to use those methods as if they were defined locally.

To create objects from classes, we use the `new` form:

```
(define nemo (new fish [name "Nemo"]))
(define dory (new greedy-fish [name "Dory"]))
(define monstro (new (make-greedy noisy-fish) [name "Monstro"]))
```

Names and values for initialization arguments are paired. Here, we create a regular fish called "Nemo" and a greedy fish called "Dory". To illustrate the flexibility of `new`, we also create an object from the result of mixing the `eat-until-full` method into the `noisy-fish` class and name it "Monstro".

Now that we have some objects, we can manipulate them:

```
(set-field! weight nemo 3)
(send monstro eat-until-full (list nemo dory))
```

After setting the weight of `nemo` as shown, the method call to `eat-until-full` on `monstro` causes the strings "Eating Nemo", "Eating Dory", and "Burp!" to be displayed on separate lines and returns the empty list.

In addition to conventional inheritance, Racket also allows the use of Beta-style inheritance [Lehrmann Madsen et al., 1993, Ernst, 1999]. Definitions of new augmentable methods use `define/pubment` instead of `define/public`. Calls to augmentable methods access the first definition in the class hierarchy, which may choose to access augmenting behavior in subclasses via the `inner` form. Such augmenting behavior is defined using the `define/augment` keyword, the analogue to `define/override`. Goldberg et al. [2004] describe how conventional and Beta-style inheritance coexist in the Racket class system.
### 4.1.1 Contracts on Classes

Let us illustrate the language for class contracts with a contract for our *fish* classes:

```scheme
(define fish/c
  (class/c
    (init [name string?])
    (field [weight positive?])
    [eat (⇒ food/c #:pre (not (send this full?)) void?)]
    [full? (⇒ boolean?)]
    (inherit [eat (⇒ food/c void?)])))
```

According to this contract, a fish has three important features. First, a fish has a name that is provided during instantiation. Second, a fish stores its current weight. Third, a fish has a method for eating food.

In general, the `class/c` contract combinator takes a series of (possibly tagged) clauses of names and contracts. Contract clauses without a tag, such as the first occurrence of `eat`, describe how object clients may use the method. Clauses tagged by `init` check the arguments needed for constructing new objects. Clauses tagged by `field` protect public fields. Clauses tagged by `inherit` guard the use of a method by subclasses.

As this first class contract illustrates, a class contract differentiates the object interface from the specification interface [Lamping, 1993]. That is, it may impose distinct constraints on the behavior of object clients and subclasses. Calls to `eat` via `send` must occur only on fish that are not full, while subclasses may force-feed fully satisfied fish. Such preconditions may use `this` to access the target object.

One important point to note here concerns the intent of our class contracts. In an un-typed world, classes tend to serve as vessels of reusable code. Hence inheritance just mimics a module import. Consequently, our class contracts enforce implementation inheritance, *not* behavioral subclassing [Liskov and Wing, 1994]. In particular, if a subclass later extends a contracted fish by overriding the `eat` method, the contract is not enforced on the new overriding implementation. If control flow reaches the implementation in the contracted fish via a call from its subclass, however, it is checked appropriately.

Class contracts can be part of a function contract:

```scheme
(define make-greedy/c
  (⇒ (class/c (inherit [eat (⇒ food/c void?)])
       [full? (⇒ boolean?)])
    (class/c [eat-until-full (⇒ (listof food/c) void?)]
             (override [eat (⇒ food/c any/c)]))))
```
4.1. DESIGNING CLASS CONTRACTS

Here, the domain contract specifies that the superclass must contain the public methods \textit{eat} and \textit{full?} with expectations for uses by the subclass. The range contract includes an ordinary method specification and a method clause tagged by \texttt{override}. The latter describes expectations for overriding method implementations in subclasses of the function’s result. Thus, a subclass of the result may provide an overriding implementation of \textit{eat} that returns any value instead of being forced to return the \texttt{void} value.

Both \texttt{inherit} and \texttt{override} clauses deal with dynamic dispatch inside the class hierarchy, but they specify different clients and servers for a given method implementation. Contracts in \texttt{inherit} clauses take effect when the contracted class provides a method implementation that is called within a subclass. That is, the contracted class is the server and the subclass is the client. With \texttt{override} contracts, the information flow is reversed; the subclass provides the method implementation while the protected class contains the call site. Thus, the subclass is the server and the protected class is the client. In other words, \texttt{override} clauses are particularly useful in the context of the template-and-hook pattern.

If class contracts worked like function contracts, then the range contract above would restrict clients of the subclass to two methods: \textit{eat-until-full} and \textit{eat}. Instead, class contracts are \textit{translucent}, i.e., they allow unlisted class features to be used or extended without constraints. For example, the \texttt{full?} and \texttt{eat} methods of the result, inherited from the superclass, are still accessible even though they are not mentioned in the range contract.

So far our contracts describe the available features of a class, but we must also have a method for describing which features are absent. For example, defining a method that already exists in a superclass is an error in Racket’s class system. This error is triggered if a mixin is erroneously applied to a class that already defines the mixin’s new functionality. For example, we want to keep the \texttt{make-greedy} mixin from being applied to a \texttt{fish} class that already contains the \texttt{eat-until-full} method, and we wish to express this constraint in class contracts [Bracha, 1992].

In response, we add an \texttt{absent} clause to our class contracts that lists fields and methods that should \textit{not} be present in the contracted class. For example, here is a revised contract for \texttt{make-greedy}:

\begin{verbatim}
(define make-greedy/c
  (→ (class/c (absent eat-until-full)
           (inherit [eat (⇒ food/c void?)])
           [full? (⇒ boolean?)])
    (class/c [eat-until-full (⇒ (listof food/c) void?)]
             (override [eat (⇒ food/c any/c)]))))
\end{verbatim}
The **absent** clause specifies that the argument class should not define a public method *eat-until-full*. If it does, the contract system signals a contract violation with the appropriate blame as soon as the mixin is applied.

Because Racket’s class system supports augmentable methods, our contract system can describe those too. An **augment** contract clause checks that the method is augmentable in the class and describes how subclasses may call the original implementation via dynamic dispatch. Restrictions on augmenting behavior in subclasses are specified using the **inner** contract clause. That is, the **augment** and **inner** clauses are analogues to the **inherit** and **override** clauses for traditional inheritance.

### 4.1.2 Contracts, Blame, and Dynamic Dispatch

Our class contracts protect implementations, because a class contract does not automatically propagate to subclasses. Only those contracts added to the class hierarchy between the use of a method and the accessed implementation take effect. Given dynamic dispatch, we cannot determine which method implementation is accessed until runtime for a given call site. Therefore, determining which contracts apply, and thus which parties to blame, cannot be done statically in our system.

To illustrate this issue, we first define `checked-greedy`, which is the same as `make-greedy` but checked with the `make-greedy/c` contract:

```racket
(define/contract checked-greedy make-greedy/c make-greedy)
(define greedy (checked-greedy fish))
```

We also create a subclass of `greedy` that overrides the definition of `eat`:

```racket
(define nibbler
  (class greedy
    (super-new)
    (inherit-field weight)
    (define/override (eat f)
      (when (> (get-field weight f) 1)
        (set! weight (+ weight 1))
        (set-field! weight f (- (get-field weight f) 1))))
    f))

(define marlin (new nibbler [name "Marlin"]))
```

A call to `eat` on `marlin` returns the partially-eaten food, taking advantage of the relaxed condition on the result of `eat` as described in the **override** clause.
**4.2. THE MODEL**

Let $dom$ be the domain contract of $\text{make-greedy/c}$ and $rng$ be the range contract. Assuming that the three class definitions exist in distinct contract regions, the contract boundaries between $\text{fish}$, $\text{greedy}$, and $\text{nibbler}$ are as follows:

![Diagram showing contract boundaries between fish, greedy, and nibbler]

That is, $dom$ mediates interactions between $\text{fish}$ and its subclasses, and $rng$ mediates interactions between $\text{nibbler}$ and its superclasses.

If we call $\text{eat-until-full}$ on $\text{dory}$, a $\text{greedy}$ fish:

\[
(\text{send } \text{dory } \text{eat-until-full} \ldots)
\]

then the call to $\text{eat}$ inside $\text{greedy}$ uses the implementation from the superclass $\text{fish}$. The call is therefore checked according to the $\text{inherit}$ clause in $dom$, because the call crosses the contract boundary between $\text{fish}$ and $\text{greedy}$. If we instead call $\text{eat-until-full}$ on $\text{marlin}$, a $\text{nibbler}$:

\[
(\text{send } \text{marlin } \text{eat-until-full} \ldots)
\]

then the implementation of $\text{eat}$ comes from $\text{nibbler}$. This call is checked according to the $\text{override}$ clause in $rng$, since the call crosses the contract boundary between $\text{greedy}$ and $\text{nibbler}$, and the non-void return from $\text{eat}$ in $\text{nibbler}$ is allowed. In short, we get two different contract stories, depending on which particular class is being instantiated.

Since we do not know which implementation is invoked from a call due to dynamic dispatch, we do not know which contracts are enforced until runtime. Since we do not know which contracts are enforced until runtime, we cannot statically determine the parties to blame for a given method call. Thus, blame assignment for contract violations within the class hierarchy is even more complex than that for higher-order functions, and getting it right calls for a manageable mathematical model.

### 4.2 The Model

The most fundamental question about a new contract system is whether it assigns blame correctly. To give a satisfying answer, we must formally define and prove blame correctness. We do so on the basis of a formal model that carries enough information to describe blame correctness [Dimoulas et al., 2011]. The idea is that a contract specifies constraints on the interaction between a component that provides a service\(^2\) and its clients. Each party

\(^2\)We use the term service loosely as an umbrella for any possible export from a component.
involved in a contract, the server or a client, is responsible for meeting only some parts of the contract, dubbed its obligations. When a party exchanges values with its contract partners, the values are checked against some of the obligations of the party. If the values do not meet the specifications described by the obligations, the party is blamed for violating the contract.

**Informal Def. 4.2.1 (Blame Correctness).** A contract system assigns blame correctly for a contract violation if it blames the party that provides the value that falsifies one of the party’s obligations.

CPCF is the starting point for our formalization. It extends Plotkin’s PCF [Plotkin, 1977] with contracts for higher-order functions. We gradually add classes and other constructs to approximate the essence of Racket.

**Figure 4.1 CPCF syntax**

\[
\begin{align*}
\text{Contracts} & : \kappa ::= \text{any} \mid \text{flat}(e) \mid \kappa_{1} \ldots \rightarrow \kappa \\
\text{Terms} & : e ::= v \mid x \mid e \cdot e \ldots \mid e + e \mid e - e \mid e \land e \mid e \lor e \\
& \quad \mid \text{zero?}(e) \mid \text{if} e \ e \ e \mid \text{mon}_{l,k}^{l,k}(\kappa, e) \mid \text{error}^{l} \\
\text{Values} & : v ::= 0 \mid 1 \mid -1 \mid \ldots \mid \lambda(x) . . . e \mid \text{tt} \mid \text{ff}
\end{align*}
\]

### 4.2.1 Contract PCF

Figure 4.1 presents the syntax of CPCF. The language comes with three kinds of contracts \(\kappa\): any, which never fails; \(\text{flat}(e)\), first-order checks on base values; or \(\kappa_{1} \ldots \rightarrow \kappa_{r}\), higher-order contracts on functions. A programmer can protect a term \(e\) with a contract \(\kappa\) with the monitor construct \(\text{mon}_{l,k}^{l,k}(\kappa, e)\). Such a monitor divides the program into two components: the term \(e\), dubbed the server, and the context of \(\text{mon}_{l,k}^{l,k}(\kappa, e)\), dubbed the client. The monitor regulates the interaction of the two components, which are also called the parties of the contract. The labels \(l\) and \(k\) on the monitor are identifiers for the server, and client, respectively. As Dimoulas et al. [2011] show, a contract in a higher-order world deserves its own label; we use \(j\) for this purpose. Since such contracts call unknown code with unforeseeable consequences, a contract itself may break invariants and deserve blame. The three identifiers are used to report violations.

The obligations of each party are a subset of the flat pieces of the contract. The server obligations consist of the flat pieces in positive positions in the contract while the client obligations correspond to those in negative positions. The any contract never fails and thus imposes no obligations.
4.2. THE MODEL

We also say that each party owns the terms and values it contains. As the two parties interact and exchange values, the values change ownership status. Tracing ownership is necessary to state and prove that only contributing parties are blamed. CPCF uses ownership annotations \( \| e \|^I \) to designate that a party \( l \) owns a term \( e \), and obligation annotations \([ \text{flat}(e_c) ]^I \) to mark that a party \( l \) is responsible for a flat contract \( \text{flat}(e_c) \).

Figure 4.2 CPCF syntax with Ownership and Obligations

| Contracts  | \( \kappa ::= \ldots \mid [ \text{flat}(e) ]^I \) |
| Terms      | \( e ::= \ldots \mid \| e \|^I \) |
| Values     | \( v ::= \ldots \mid \| v \|^I \) |

4.2.2 Adding First-Class Classes to CPCF

The result of adding first-class classes and their contracts to CPCF is CFCC; see figure 4.3. CFCC comes with first-class classes and objects. These classes feature public, inherited, and overridden methods, and CFCC offers contracts for all of these features. However, classes do not have fields, and our model does not account for Beta-style inheritance. Furthermore, it does not differentiate between contracts on inherited and public methods. These omissions, though, do not reduce the value of our theoretical validation. Since the traditional purpose of a model is to distill the essence of a language in order to make a concise and convincing argument about the correctness of its design, our theoretical model is intentionally smaller than the Racket class system. We make sure, however, that it includes the features of the class system that pose the greatest challenges to the framework of Dimoulas et al. [2011]: inheritance and overridden methods.\(^3\)

In this setting, classes consist of the superclass expression, the identifiers of inherited methods, and the definitions of public and overridden methods. Class values contain the superclass value and definitions for all the methods defined locally in the class. In addition, each class value comes with a dynamically generated unique identifier \( i \). The inclusion of the superclass inside a class value makes it easy to track the class hierarchy in a setting with first-class classes. The root of the class hierarchy is the class value \( \text{object}\% \). Because of the omission of fields from the model, an object of a class is just a wrapper around the class value.

A method \( m \) in object \( e_o \) can be called in one of three ways: either directly via \( \text{send}(e_o, m, e_1 \ldots) \); internally from the class where the call occurs via \( \text{isend}^d(e_o, m, e_1 \ldots) \), which corresponds to the local use of class functions; or via a super call, \( \text{super}^d(e_o, m, e_1 \ldots) \).

\(^3\) One may argue that our contract system could be derived through an encoding in an even smaller object calculus, but this would obscure the relation between ownership and flow tracking in the model.
The latter two forms are annotated with the unique identifier \( \iota \) of the class value where the call occurs. The identifier is used to locate the class from which the call is performed, starting from the class of the object \( e_o \) and proceeding through the class hierarchy via superclasses. Since the unique identifier of a class value is not accessible to the programmer, all super and internal calls in the source are annotated with the default class identifier \( \triangleright \).

CFCC features a special form of contracts for classes and objects:

\[
\text{class}/c\{ \text{public } [m_{p_1} (\kappa_1 \kappa_{p_1} \ldots \rightarrow \kappa_{p_1})] \ldots \text{override } [m_{o_1} (\kappa_{o_1} \kappa_{1} \ldots \rightarrow \kappa_{o_1})] \ldots \}
\]

Public method contracts aggregate the public and inherited contracts of section 4.1.1. Contracts for overridden methods, though, are client side contracts. They differ significantly from public and inherited contracts, and thus the model keeps them separate from public method specifications.

In order to keep track of the contracts imposed on a class or an object, we wrap the class and the object in a guard. It contains the protected class or object, the contracts and the labels for the server, client and contract. Guards are a specialized form of monitor, but, in contrast to monitors, are treated as values.

<table>
<thead>
<tr>
<th>Figure 4.3 CFCC syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contracts</strong></td>
</tr>
<tr>
<td><strong>Terms</strong></td>
</tr>
<tr>
<td><strong>Values</strong></td>
</tr>
<tr>
<td><strong>Class Values</strong></td>
</tr>
<tr>
<td><strong>Classes</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Objects</strong></td>
</tr>
<tr>
<td><strong>Guards</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

### 4.2.3 Well-Formed Programs

Like CPCF, CFCC comes with annotations for ownership and obligations. Not all annotations are meaningful in the source code, however. Specifically, ownership annotations must coincide with component boundaries. In the source code, they must agree with the labels on monitors. We express these constraints using the judgment \( l \vdash e \). It states that term \( e \) is well-formed under owner \( l \). A well-formed program \( e \) is a closed term such that \( l_o \vdash e \).
where \( l_o \) denotes ownership for the complete program. Figure 4.4 lists the complete set of rules. The most interesting rule concerns monitor expressions:

\[
\frac{k \vdash e \quad k; l; j \triangleright \kappa}{l \vdash \text{mon}_{j,l}^k(k, \| e \|^k)}
\]

A monitor term is well-formed if the owner of the term is the client party of the contract and the server party is the owner of the guarded term. In addition, the contract must be well-formed according to the judgment \( k; l; j \triangleright \kappa \).

The rules for well-formed contracts in figure 4.5 check if the client and server labels on the monitor are distributed on the negative and positive flat contracts of \( \kappa \) correspondingly. Furthermore, the rules demand that the owner of the predicates in all the flat contracts of \( \kappa \) is the contract \( j \). The rule for the \textbf{any} contract has no premises as the \textbf{any} contract imposes no obligations on the contract parties. The rule for class contracts flips the client and server labels to check the overridden method contracts. The reason for flipping the labels is that these contracts are client side contracts.

Also, note that the rules for well-formed terms exclude intermediate terms such as errors, guards, class values, and objects and force all class identifiers on internal calls and super calls to be the default class identifier. \( \Box \)

Figure 4.4 Ownership coincides with contract monitors

\begin{align*}
\frac{l \vdash e}{n \in \text{Integers}} & \quad \frac{l \vdash \text{tt}}{l \vdash \text{ff}} & \quad \frac{l \vdash e_1}{l \vdash \text{zero}(e_1)} \\
\frac{l \vdash e_1}{l \vdash e_1 + e_2} & \quad \frac{l \vdash e_1}{l \vdash e_1 - e_2} & \quad \frac{l \vdash e_1}{l \vdash e_1 \land e_2} & \quad \frac{l \vdash e_1}{l \vdash e_1 \lor e_2} \\
\frac{l \vdash x}{l \vdash \lambda(x_1 \ldots, e)} & \quad \frac{l \vdash e_f}{l \vdash e_1 \ldots} & \quad k \vdash e & \quad k; l; j \triangleright \kappa \\
\frac{l \vdash e_1}{l \vdash \text{if} e_1 e_2 e_3} & \quad \frac{l \vdash e_1}{l \vdash \text{object}\%} & \quad \frac{l \vdash e_o}{l \vdash \text{super}(e_o, m, e_1 \ldots)} \\
\frac{l \vdash \text{new}(e)}{l \vdash \text{send}(e_o, m, e_1 \ldots)} & \quad \frac{l \vdash e_1}{l \vdash \text{isend}(e_o, m, e_1 \ldots)} & \frac{l \vdash e_1}{l \vdash \text{class}\{ e_s, e_p_1 \ldots, e_o_1 \ldots \}} \\
\frac{k \vdash e}{l \vdash \| e \|^k} & \quad \frac{l \vdash e_s}{l \vdash \text{inherit} m_i \ldots} & \frac{l \vdash \text{public}[m_p_1 \{ \text{this}_p_1, x_1^{p_1} \ldots \} e_p_1 \ldots]}{l \vdash \text{override}[m_o_1 \{ \text{this}_o_1, x_1^{o_1} \ldots \} e_o_1 \ldots]}
\end{align*}
4.2.4 CFCC Semantics

We next equip CFCC with a reduction semantics [Felleisen et al., 2009a]. Each reduction rule deterministically decomposes a program into an evaluation context and a redex. A single-step reduction relation specifies how a redex is transformed in a single computation step. The closure of the single-step reduction over evaluation contexts gives us the compatible closure. The transitive closure of this relation is the reduction relation of the language, which defines the evaluator of the language.

The first step towards defining the reduction relation for CFCC is to equip the language with evaluation contexts. The grammar of evaluation contexts in figure 4.6 is parameterized over the owner of the hole. In particular, in an evaluation context $E^l$, label $l$ corresponds to the ownership annotation or the server label of the monitor closest to the hole. Given
certain initial conditions, we show in section 4.3 that the two always coincide. We use $E^{\ell_o}$ to indicate an evaluation context that contains no ownership annotations or monitors on the path to the hole. In this case, the hole belongs to the owner of the entire program. The label parameter plays a key role in keeping track of ownership during reduction.

Figure 4.7 Reduction semantics for CFCC

\[
E'[\ldots] \quad \mapsto E'[\ldots]
\]

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$|n_1|^k + |n_2|^l$</td>
<td>$n$ where $n_1 + n_2 = n$</td>
</tr>
<tr>
<td>$|n_1|^k - |n_2|^l$</td>
<td>$n$ where $n_1 - n_2 = n$</td>
</tr>
<tr>
<td>$\text{zero}'(|0|^k)$</td>
<td>$\text{tt}$</td>
</tr>
<tr>
<td>$\text{zero}'(|v|^k)$</td>
<td>$\text{ff}$ if $n \neq 0$</td>
</tr>
<tr>
<td>$|v_1|^k \land |v_2|^l$</td>
<td>$v$ where $v_1 \land v_2 = v$</td>
</tr>
<tr>
<td>$|v_1|^k \lor |v_2|^l$</td>
<td>$v$ where $v_1 \lor v_2 = v$</td>
</tr>
<tr>
<td>$\text{if} \ |\text{tt}|^k \ \text{e}_1 \ \text{e}_2$</td>
<td>$\text{e}_1$</td>
</tr>
<tr>
<td>$\text{if} \ |\text{ff}|^k \ \text{e}_1 \ \text{e}_2$</td>
<td>$\text{e}_2$</td>
</tr>
<tr>
<td>$|\lambda(x_1_\ldots_\ldots _x_n).e|^k \ /_l v_1 \ldots v_n$</td>
<td>$|{|v_1|^k \ldots |v_n|^k /_l x_1 \ldots x_n}e|^k$</td>
</tr>
<tr>
<td>$\text{new}(|\gamma|^k)$</td>
<td>$\text{object}(|\gamma|^k)$</td>
</tr>
<tr>
<td>$\text{class}{|\gamma|^k \ \text{m}_1 \ldots }$</td>
<td>$\text{class/'{|\gamma|^k \ \text{m}_1 \ldots }$</td>
</tr>
<tr>
<td>$\text{public} \ [m_{p_1}(\text{this}_{p_1} \ \text{x}_1 \ldots \text{x}_n)<em>1 \ldots m</em>{p_1}]$</td>
<td>$[m_{p_1}(\text{this}_{p_1} \ \text{x}_1 \ldots \text{x}_n)<em>1 \ldots m</em>{p_1}]$</td>
</tr>
<tr>
<td>$\text{override} \ [m_{o_1}(\text{this}_{o_1} \ \text{x}_1 \ldots \text{x}_n)<em>1 \ldots m</em>{o_1}]$</td>
<td>$[m_{o_1}(\text{this}_{o_1} \ \text{x}_1 \ldots \text{x}_n)<em>1 \ldots m</em>{o_1}]$</td>
</tr>
<tr>
<td>\hspace{2em} where $\ell$ fresh (does not occur in the eval. context filled with the entire reduc)</td>
<td></td>
</tr>
<tr>
<td>if $\mathcal{CV}[|\gamma|^k]$ and $\forall j, \text{m}_j \in \text{Methods}[|\gamma|^k]$</td>
<td></td>
</tr>
<tr>
<td>and $\forall j, \text{m}<em>j \in \text{Methods}[|\gamma|^k]$ and $\forall j, \text{m}</em>{p_j} \notin \text{Methods}[|\gamma|^k]$</td>
<td></td>
</tr>
<tr>
<td>$\text{send}(|\gamma|^k, v, _\ldots _)$</td>
<td>$\ell {|\gamma|^k \ \text{v} \ldots }$ where $\ell = \text{Pull}[|\gamma|^k, m]$</td>
</tr>
<tr>
<td>if $\text{Object}[|\gamma|^k]$ and $m \in \text{Methods}[|\gamma|^k]$</td>
<td></td>
</tr>
<tr>
<td>$\text{super}^k(|\gamma|^k, v, _\ldots _)$</td>
<td>$\ell {|\gamma|^k \ \text{v} \ldots }$ where $\ell = \text{Pull}[|\gamma|^k, \ell, m]$</td>
</tr>
<tr>
<td>if $\text{Object}[|\gamma|^k]$ and $\mathcal{L}[|\gamma|^k, \ell, m]$ and $m \in \text{Methods}[|\mathcal{L}[|\gamma|^k, \ell, \ell, m]]</td>
<td>$</td>
</tr>
<tr>
<td>$\text{isend}(|\gamma|^k, v, _\ldots _)$</td>
<td>$\ell {|\gamma|^k \ \text{v} \ldots }$ where $\ell = \text{Find}[|\gamma|^k, \ell, m, \ell]$</td>
</tr>
<tr>
<td>if $\text{Object}[|\gamma|^k]$ and $\mathcal{L}[|\gamma|^k, \ell, m]$ and $m \in \text{Methods}[|\mathcal{L}[|\gamma|^k, \ell, \ell, m]]</td>
<td>$</td>
</tr>
<tr>
<td>$\text{mon}^k_{\ell}(\text{c}_1 \ldots \text{c}_n \mapsto \text{c}_r, v)$</td>
<td>$\lambda(x_1 \ldots x_n)$. $\text{mon}^k_{\ell}(\text{c}<em>r, v \text{mon}^k</em>{\ell}(\text{c}<em>1, x_1) \ldots \text{mon}^k</em>{\ell}(\text{c}_n, x_n))$</td>
</tr>
<tr>
<td>$\text{mon}^k_{\ell}(\text{any}, v)$</td>
<td>$v$</td>
</tr>
<tr>
<td>$\text{mon}^k_{\ell}(\text{t}(\text{flat}(\text{c})), v)$</td>
<td>$\text{if} (e v) \text{v error}^k$</td>
</tr>
<tr>
<td>$\text{mon}^k_{\ell}(|\gamma|^k)$</td>
<td>$\text{G} {|\gamma|^k }$</td>
</tr>
<tr>
<td></td>
<td>\hspace{2em} public $[m_{p_1}(\text{this}_{p_1} \ \text{c}_1 \ldots \text{c}_n)<em>1 \ldots m</em>{p_1}]$</td>
</tr>
<tr>
<td></td>
<td>\hspace{2em} $\text{override} [m_{o_1}(\text{this}_{o_1} \ \text{c}_1 \ldots \text{c}_n)<em>1 \ldots m</em>{o_1}]$</td>
</tr>
<tr>
<td>\hspace{2em} where $\ell = \text{class}/\text{c}{\text{public} \ [m_{p_1}(\text{this}_{p_1} \ \text{c}_1 \ldots \text{c}_n)<em>1 \ldots m</em>{p_1}]$</td>
<td></td>
</tr>
<tr>
<td>\hspace{2em} $\text{override} [m_{o_1}(\text{this}_{o_1} \ \text{c}_1 \ldots \text{c}_n)<em>1 \ldots m</em>{o_1}]$</td>
<td></td>
</tr>
<tr>
<td>$E'[\text{error}^k]$</td>
<td>$\mapsto \text{error}^k$</td>
</tr>
</tbody>
</table>
In this paper, the reduction relation is the means not only for evaluating terms but also for appropriately propagating the ownership annotations. At first, though, we focus on the computational aspects of reduction and particularly the object-oriented features and contract checking. At the end of the section, we return to ownership propagation.

### 4.2.5 CFCC Object-Oriented Semantics

Figure 4.7 presents the reduction rules for CFCC. The reductions for the object-oriented constructs of the language depend on a number of meta-functions. We briefly discuss them as we describe the rules, but here we provide only the definitions of the most interesting ones.

A new object of a class can be created using the `new(e)` construct if the argument evaluates to a class value. The `CV` meta-function traverses its argument, ignoring any guard layers, until it finds a class value. If it runs into another kind of value than a class value or a guard, it rejects the argument.

If the superclass expression of a class has been evaluated to a class value, we can reduce the class expression to a class value—assuming that the inherited and overridden methods of the class are implemented in the class hierarchy to which the class belongs and that public methods are not implemented in a superclass. The `Methods` meta-function collects all the method names in a class hierarchy starting from the given class value or object. The reduction rule produces a fresh class identifier for the new class value. The special substitution function `{τ/id◆}e` replaces all occurrences of ◆ annotations on super and internal calls in e with τ.

The `send(e_o,m,e_1...)` expression performs a method call to an object if e_o evaluates to an object v. A value v is an object if it is constructed using the `object(cv)` constructor and cv is a class value or if v is a guard around an object. The reduction employs the meta-function `Pull` from figure 4.8. This meta-function traverses the class value of the object until it discovers the first definition of m. It then pulls m back to the surface, wrapping it with any contracts for m between the definition and the call site.

A super call `super◆(e_o,m,e_1...)` is performed if e_o evaluates to an object o and m is implemented in the class hierarchy that starts from the class value cv, the superclass of the class where the call site occurs. We use the `GetS` meta-function to obtain cv. This meta-function traverses o until it runs into a class value with class identifier τ and extract its superclass cv. To make sure that such a class value exists when the reduction fires, the side-conditions of the rule employs the meta-function `Is`. After obtaining cv, the rule applies `Pull` to get the contracted implementation of m.
functions on the guarded value. If the test succeeds, the value is returned. Otherwise, a contract
itors of flat contracts, Mon-
4.2.6 CFFC Contract Semantics

Internal calls have the shape `isend^e(e_o, m, e_1 ...)` and must satisfy similar constraints
as super calls. The difference is that `m` must be accessible from the class value `cv` where
the call site occurs, instead of from its superclass. The meta-function `Get` retrieves `cv`, and
the meta-function `Find` from figure 4.9 constructs the result of the reduction. The latter
meta-function ascends the class hierarchy starting from the given object until it either finds
`cv` or the first implementation of `m`. If it finds `cv` first, then it returns the result of using
`Pull` on `cv` and `m`. Otherwise, it delegates the task to `Push` from figure 4.10. This last
meta-function pushes the implementation of `m` up the class hierarchy until it reaches `cv`
and then returns the result. The implementation of `m` initially detected is an overridden
implementation of the one reachable from `cv`. Thus, the implementation of `m` is wrapped
with any `override` contracts encountered en route to `cv`.

4.2. THE MODEL

Figure 4.8 The `Pull` meta-function

\[
\begin{align*}
Pull[^\gamma^k, m] &= \|\lambda(this \, x_1 ...) \cdot e^k \| \\
\text{if } [m(this \, x_1 \ldots) \cdot e] \in \{[m_1(this \, x_1 \ldots) \cdot e_1] \ldots\} \\
\text{where } \gamma &= \text{class/\gamma' \{ } ||\gamma'^{\gamma'} \text{ methods } [m_1(this \, x_1 \ldots) \cdot e_1] \ldots\}
\end{align*}
\]

\[
\begin{align*}
Pull[^\gamma^k, m] &= ||Pull[^\gamma'^{\gamma'}, m]||^k \\
\text{if } m \notin \{m_1, \ldots\} \\
\text{where } \gamma &= \text{class/\gamma' \{ } ||\gamma'^{\gamma'} \text{ methods } [m_1(this \, x_1 \ldots) \cdot e_1] \ldots\}
\end{align*}
\]

\[
\begin{align*}
Pull[^\gamma^k, m] &= ||Pull[^\gamma', m]||^k \\
\text{if } m \notin \{m_1, \ldots\} \\
\text{where } \gamma &= \text{Get[ } ||\gamma' \text{ methods } [m_1 \ldots] \ldots\}
\end{align*}
\]

\[
\begin{align*}
Pull[^\gamma^k, m] &= ||Pull[^\gamma', m]||^k \\
\text{if } m \notin \{m_1, \ldots\} \\
\text{where } \gamma &= \text{Get[ } ||\gamma' \text{ methods } [m_1 \ldots] \ldots\}
\end{align*}
\]

Contract checking in CFCC proceeds along the lines of Findler and Felleisen [2002]. Mon-
itors of flat contracts, `flat(e)`, are expanded to `if`-statements that check the predicate `e`
on the guarded value. If the test succeeds, the value is returned. Otherwise, a contract
error is raised indicating that the server party `l` broke contract `j`. Monitors of contracts
for functions \(\kappa_1 \ldots \kappa_n \mapsto \kappa_r\) are expanded into a function that first wraps each argument
Figure 4.9  The \texttt{Find} meta-function

\[
\text{Find}[[\gamma] T, t, m, l] = \text{Pull}[[\gamma] T, m]
\]
where \( \gamma = \text{class} / \text{'} \{ ||\gamma|| T \text{ methods } [m_1(\text{this}_1 x_1 \ldots) e_1] \ldots \}

\[
\text{Find}[[\gamma] T, t, m, l] = \text{Push}[[\gamma] T, ||\lambda(\text{this}_j x_1 \ldots) e_j ||(\supset k) T, t, m, l \otimes k]
\]
if \([m(\text{this}_1 x_1 \ldots) e] \in \{ [m_1(\text{this}_1 x_1 \ldots) e_1] \ldots \}
where \( \gamma = \text{class} / \text{'} \{ ||\gamma|| T \text{ methods } [m_1(\text{this}_1 x_1 \ldots) e_1] \ldots \}

\[
\text{Find}[[\gamma] T, t, m, l] = \text{Find}[[\gamma] T, t, m, l \otimes k]
\]
if \( m \notin \{m_1 \ldots \}
where \( \gamma = \text{class} / \text{'} \{ ||\gamma|| T \text{ methods } [m_1(\text{this}_1 x_1 \ldots) e_1] \ldots \}

\[
\text{Find}[[\gamma] T, t, m, l] = \text{Find}[[\gamma] T, t, m, l \otimes k]
\]
where \( \gamma = \text{G} \{ ||\gamma|| T \}
\[
\begin{align*}
\text{public} & [m_{p_1} (\kappa_{\text{this}}^{\rho_{p_1}} k_{1}^{\delta_1} \ldots \mapsto \kappa_{\text{this}}^{\rho_{p_1}} k_{j}^{\delta_j}) \ldots ] \\
\text{override} & [m_{o_1} (\kappa_{\text{this}}^{\rho_{o_1}} k_{1}^{\delta_1} \ldots \mapsto \kappa_{\text{this}}^{\rho_{o_1}} k_{j}^{\delta_j}) \ldots ]
\end{align*}
\]

Figure 4.10  The \texttt{Push} meta-function

\[
\text{Push}[[\gamma] T, v, t, m, l] = v
\]
where \( \gamma = \text{class} / \text{'} \{ ||\gamma|| T \text{ methods } [m_1(\text{this}_1 x_1 \ldots) e_1] \ldots \}

\[
\text{Push}[[\gamma] T, v, t, m, l] = \text{Push}[[\gamma] T, ||v|| T, t, m, l \otimes k]
\]
where \( \gamma = \text{class} / \text{'} \{ ||\gamma|| T \text{ methods } [m_1(\text{this}_1 x_1 \ldots) e_1] \ldots \}

\[
\text{Push}[[\gamma] T, v, t, m, l] = \text{Push}[[\gamma] T, ||v|| T, t, m, l \otimes k]
\]
if \( m \notin \{m_1 \ldots \}
where \( \gamma = \text{G} \{ ||\gamma|| T \}
\[
\begin{align*}
\text{public} & [m_{p_1} (\kappa_{\text{this}}^{\rho_{p_1}} k_{1}^{\delta_1} \ldots \mapsto \kappa_{\text{this}}^{\rho_{p_1}} k_{j}^{\delta_j}) \ldots ] \\
\text{override} & [m_{o_1} (\kappa_{\text{this}}^{\rho_{o_1}} k_{1}^{\delta_1} \ldots \mapsto \kappa_{\text{this}}^{\rho_{o_1}} k_{j}^{\delta_j}) \ldots ]
\end{align*}
\]

\[
\text{Push}[[\gamma] T, v, t, m, l] = \text{Push}[[\gamma] T, ||\text{mon}^{l, (\kappa_{\text{this}} k_1 \ldots \mapsto k_{r_1})} T, t, m, l \otimes k]
\]
if \([m (\kappa_{\text{this}} k_1 \ldots \mapsto k_{r_1})] \in \{ [m_1 (\kappa_{\text{this}} k_1 \ldots \mapsto k_{r_1})] \ldots \}
where \( \gamma = \text{G} \{ ||\gamma|| T \}
\[
\begin{align*}
\text{public} & [m_{p_1} (\kappa_{\text{this}}^{\rho_{p_1}} k_{1}^{\delta_1} \ldots \mapsto \kappa_{\text{this}}^{\rho_{p_1}} k_{j}^{\delta_j}) \ldots ] \\
\text{override} & [m_{o_1} (\kappa_{\text{this}}^{\rho_{o_1}} k_{1}^{\delta_1} \ldots \mapsto \kappa_{\text{this}}^{\rho_{o_1}} k_{j}^{\delta_j}) \ldots ]
\end{align*}
\]
with a monitor of the corresponding precondition contract $\kappa_i$ and then applies the monitored arguments to the original function. The result of the application is guarded with the postcondition contract $\kappa_r$. The monitors for the arguments flip the labels for server and client, as the client is responsible for the positive pieces of the contract on the argument. This process delays contract checking until there are witnesses to check all the flat pieces of a function contract.

Monitoring contracts for classes and objects is also delayed. Attaching a contract to a class value or an object results in a guard that wraps the value and contains the method contract. Moreover, each contract is annotated with the server, client and contract labels of the monitor. For the public method contracts, the labels are used as-is while the server and client labels are flipped for the overridden method contracts. This reflects that the latter are client-side contracts rather than server-side ones.

### 4.2.7 CFCC Ownership Semantics

Let us now return to ownership propagation. The initial owner of a value is its creator. As values pass from one component to another during evaluation, they collect stacks of ownership annotations:

$$\|e\|^{l_n} = \|\ldots\|e\|^{l_1}\ldots\|^{l_n}$$

$$\|e\|^{l_n} = \|\ldots\|e\|^{l_n}\ldots\|^{l_1}$$

We drop the subscript $n$ when the height of the stack is irrelevant. Furthermore, we define the following operations on ownership stacks:

$$k \odot l_n = \begin{cases} l_1 & \text{if } l_n = l_1l_2\ldots l_n \\ k & \text{otherwise} \end{cases}$$

$$k \oplus l_n = \begin{cases} l_n & \text{if } l_n = l_1l_2\ldots l_n \\ k & \text{otherwise} \end{cases}$$

The $\beta_v$ reduction is the primary means for values to migrate from one component to another, and thus it is the most intriguing rule with respect to ownership:

$$E'\left[\lambda(x_1\ldots x_n).e\| v_1\ldots v_n\right] \mapsto E'\left[\|v_1\|^{l_1}_T\ldots\|v_n\|^{l_n}_T / x_1\ldots x_n\} e\|^{l_T}_T\right]$$

The owner $l$ of the hole is also the owner of the arguments $v_1\ldots v_n$. The function may have a different set of owners $\bar{l}$. Thus the arguments are first tagged as properties of the context, $\|v_1\|^{l'}\ldots\|v_n\|^{l'}$, and are then passed to the function; this means they also pick up ownership tags from all the owners of the function. The order of the labels is the order
in which the arguments encounter the owners of the function. Finally, the arguments are substituted for the function parameters in the body of the function. The resulting term has the same owners as the function.

Ownership is also involved in the definitions of the meta-functions. As terms are pulled or pushed, they pick up any ownership tags they meet. For instance in the definition of \( \text{Pull} \), the method implementation records the migration path of its journey. For \( \text{Find} \) and \( \text{Push} \), we use a more complex accumulator. If we run into the target method implementation first, the direction of the movement of the method implementation is towards the superclass. Thus, the method gets further away from the calling context. However, when we locate the superclass, we install the method back in the calling context. In order to make sure that the method remains well-formed no matter where it emerges, we equip \( \text{Find} \) with an argument that accumulates the owners of the class tables. Upon retrieval of the method implementation, we tag it with the owner label before passing it to the superclass. Since the method is well-formed in its original location, the ownership annotation ensures that it remains so in any context. The extra label argument serves a similar purpose for \( \text{GetS} \) and \( \text{Get} \).

To sum up, the creator of a value is its initial owner but values accumulate owners as they cross component boundaries through function application or method invocation. This accumulation requires somewhat complex machinery to keep track of the flow of inherited or overridden methods. As the definition of our meta-functions demonstrate, however, this can be done in a rather intuitive way while unearthing insights about the migration of method implementations between classes and objects. The key is that ownership tracking and contract checking are \textit{not} intertwined, which allows us to discuss the correctness of blame assignment in a way that circumvents the implementation of contract monitors for higher-order features. That is, when the system checks a base value against a flat contract, the owner of the value (ownership) should be the component responsible for the flat contract (obligation) and the component that is blamed if the contract fails (blame label bookkeeping by the contract system).

### 4.3 Blame Correctness

Ownership and obligation annotations help us formulate a blame correctness criterion.

**Definition 4.3.1 (Blame Correctness).** A contract system is blame correct if for all terms \( e_0 \) such that \( l_o \vdash e_0 \) and \( e_0 \rightarrow\rightarrow E[l_{\text{mon}}k,\dagger([\text{flat}(e_1)]^\parallel, \|v_1\|)], v_1 = \|v\|^k \) and \( k \in \bar{l} \). \textbf{Note:} The identity of the \( \dagger \) labels is irrelevant.
That is, a contract system is defined to be correct if every time we check a value against a flat contract, the server label on the monitor is the owner of the value and the flat contract is part of the server’s obligations. This implies that if the contract check fails, the blamed party is the one that contributed the invalid value and the one that is also responsible for meeting the violated contract. This formal definition translates the informal definition 4.2.1 at the beginning of section 4.2 into a logical statement about reduction systems that can be (in)validated.

The proof of blame correctness for a contract system uses a variant of the standard subject reduction strategy. The first step is to develop the appropriate subject. We observe that the well-formedness judgments $l \vdash e$ and $k; l; j \triangleright \kappa$ imply the desired property for monitors of flat contracts. Unfortunately, the reduction process generates expressions that do not preserve these judgments. However, this mismatch is only temporary; after a few reduction steps, the expression becomes well-formed again. To cope with this detour, we devise a generalization of the well-formed judgments for terms and contracts that is loose enough to describe a program at each point in the reduction trace and tight enough to imply blame correctness. The generalization and the proof of blame correctness can be found in appendix A.

### 4.4 Implementation

While our mathematical model helps us prove basic properties, it does not quite explain how to implement contracts for first-class classes. Following Findler and Blume [2006], we explain our implementation of contracts as pairs of mathematical projection functions, one for each contract party. For first-order values, these projections return the value or signal an error; for higher-order values, they return similar values restricted to good behavior from the client/server perspective.

Thus, $\text{class}/c$ denotes a function that maps the two contract parties into a pair of projections from classes to classes. At contract boundaries, the run-time system applies these projections to exported classes to obtain classes equipped with run-time monitors. One way to understand this idea is to compare the two parts of figure 4.11. The top of the figure presents a source configuration complete with classes, objects, contracts, and boundaries. In the bottom part of the figure, the dotted squares represent classes created from applying projections to classes. The inheritance arrows connect the derived classes to the originals. Of course, a complete understanding of the process requires some background concerning the implementation of classes and objects in Racket, our choice of implementation language.
4.4.1 Basics

Classes and objects are implemented⁴ as heterogeneous opaque structures. Opaqueness guarantees safety. The structure contains all the necessary information for constructing objects and looking up methods and fields. Object construction is handled by functions that create structural representations of objects using the initialization arguments and the initialization expressions of the class.

The class representation contains a vector of methods and a hash table that maps a method name into a vector index. Fields are represented in the same fashion. The class compiler transforms both a field access and a method call into computations that use the appropriate vector slots. It can compute the vector index for internal uses of fields and methods statically, whereas the index must be computed at runtime for external references.

⁴Racket’s class system is implemented via a syntax extension library [Flatt et al., 2006], though here we describe the ideas as if it were a regular compiler.
Field assignments are converted into vector assignments. Calls to `super` methods index into the method table of the super class.

In short, all method calls access the same method vector, whether they are internal calls, external sends, or `super` calls. Each kind of contract clause affects a different kind of call, however. We could store the different projections and apply the correct projection for a particular call, but applying contract projections can be a costly operation. Our implementation therefore trades space for run-time speed; that is, it separates out the method vector into three vectors, illustrated by figure 4.12. External calls via `send` use the `meth` vector, `super` calls access `smeth`, and direct internal calls go through `imeth`. The revised compiler converts each operation into an indexed access to the appropriate vector. This separation works for methods calls via `send` and `super` calls; internal calls require additional changes, which we describe in section 4.4.2.

For mutable fields, our compiler stores two pairs of setters and getters for each field in the class value. One pair protects external uses of the field through objects, and the other pair protects accesses within the class hierarchy. When a contract with a field-related clause is applied to a class, the contracted version of the class contains a new pair of functions. The new getter applies the appropriate projection with covariant blame to the result of the old getter. The new setter applies the projection with contravariant blame to the incoming value, and then passes the result to the old setter.

To protect initialization arguments, the compiler creates a contracted initialization function. The new initializer first applies the appropriate projection to each argument and then passes the contracted arguments to the original initializer.
4.4.2 Internal Dynamic Dispatch

Thus far, our object organization cannot support contracts on method calls within the class hierarchy. Only certain contracts should affect such calls, namely those at contract boundaries between the class with the call site and the class that contains the method definition. We therefore employ a vector for internal calls where each entry maps to a vector of method implementations; see figure 4.13. The separation of vector entries based on contract boundaries means that contract projections for internal dynamic dispatch can be applied eagerly when creating the contracted class value, and that we can apply different contracts on different sides of the boundary.

In addition, figure 4.13 shows that the class representation contains a table with override contracts labelled iprog. These are needed so that a future extension of the class can enforce the override contract clauses of this class.

Figure 4.14 contains a class hierarchy where one of the classes, fish, is contracted with inherit and override clauses for the method eat. Its bottom half depicts the method and projection vectors used for internal calls to eat in each class. The source class fish turns into two target classes: fish denotes the class without contracts, and fish’ depicts the
Figure 4.14 Classes with override and inherit contracts

Source configuration

```
#lang racket
(define animal
  (class object% ...
    (define/public (eat x) ...)
    (define/public (hunt) ...
      (eat ...) ...)))

(define fish
  (class animal ...
    (define/override (eat x) ...)))

(provide/contract
  [fish (class/c ...
      (inherit [eat c_i])
      (override [eat c_o])))])
```

Dispatch tables after contract application

<table>
<thead>
<tr>
<th>animal</th>
<th>fish</th>
<th>fish'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>imeth(eat)</td>
<td>imeth(eat)</td>
</tr>
<tr>
<td></td>
<td>eat_1</td>
<td>eat_2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>iproj(eat)</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>greedy</th>
<th>gobbler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>imeth(eat)</td>
</tr>
<tr>
<td></td>
<td>eat_2</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>iproj(eat)</td>
</tr>
<tr>
<td></td>
<td>p_0 1</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
</tbody>
</table>
class with contracts applied. The different method implementations are differentiated using subscripts, and primed method names denote contracted methods.

The vectors for a given method in a class contains an entry for each contract boundary between that class and the first class that defined the public method in the class hierarchy. In our example, animal and fish both use a one-element vector for method eat since there are no contract boundaries between them and animal, the first class to define eat. In contrast, the contract region introduced by provide/contract separates greedy and gobbler from fish, meaning the two classes use a vector of two elements for the eat method.

The compiler turns an internal method call into an access into the internal method table of this where the secondary index takes into account the number of contract boundaries between the current class and the first class to define the targeted method. Consider an instance of gobbler. If code using that instance calls hunt, the call from animal crosses the contract boundary to use the overridden method in greedy. Since the call in animal is compiled to retrieve the method from index 0, it uses the contracted method eat'. If the code calls gulp, the internal method call from greedy uses index 1. Thus, it retrieves eat, i.e., the uncontracted method.

As for the translation of method definitions, the compiler proceeds as follows. If a class includes a new public method, the compiler creates two one-slot vectors. The method vector contains the method code, and the projection vector contains the identity projection id. See the diagram for method eat in animal in figure 4.14.

If a subclass does not override a method, as in greedy, its method and projection vectors are the same as its superclass. If it does override the method, as in fish and gobbler, then the projection vector remains the same but the method vector is the result of applying the new method implementation to each projection from the corresponding entry in the projection vector. In gobbler, the method at index 0 is eat', the result of composing eat with the projection $p_o$, which is the implementation of contract $c_o$. The method at index 1 is just eat', since index 1 of the projection vector still contains the identity projection.

Finally, the application of a class contract to a class yields a new class that contains extended method and projection vectors. In particular, an inherit clause for a method means that the compiler wraps high-indexed entries in the method vector with appropriate contract checks until that method is overridden. Similarly, an override clause means the compiler stores the appropriate projection in low-indexed entries of the projection vector, so that they are available for creating the method vector for an overriding subclass. The creation of the method and projection vectors for fish' from those in fish follows these rules. The method in index 0 has been copied, while the method eat' at index 1 is the result of applying the contract projection for $c_i$ (from the class contract for fish) to eat'2. The
4.5 Evaluating Class Contracts

Our next task is to show that our proposed language extension is both practical and efficient. Since our implementation strategy requires significant changes to the class system, we must also ensure that existing code is not unduly affected. Hence we provide three evaluations of our system. The first describes the performance impact of the class system changes on existing uncontracted code. The second examines the utility of our design via an investigation of large libraries in our code base. The third shows that the running time of a large application that makes heavy use of the newly contracted libraries is barely affected by the new contracts.

4.5.1 Evaluating the Performance Impact on Existing Code

Our addition of class contracts involves significant changes to both the representation of classes and operations on them. As such, our implementation affects not only the performance of code that uses our contract system but also that of pre-existing code. To understand these performance effects, we measure the time needed to execute operations on classes and objects and the memory used to represent classes.

We measure time efficiency with microbenchmarks that determine the time taken for particular operations: field access, method access, instantiation, and subclassing. The experiments separate field and method access into multiple microbenchmarks where the access is either within the same class, between a class and its subclass, or by an external client via an object. For subclassing, we measure the effect of adding fields and methods separately. Each benchmark is run one hundred times, and we report the average, minimum, and maximum running times. All microbenchmarks were executed one hundred times on an Intel Core i7 860 (2.8GHz) with 8 GB of DDR3-1333 memory running Ubuntu 11.10. The “rev” column contains “port” for the old class system and “class/c” for the new class system.

Figure 4.15 presents the results. Its third column lists the number of times a microbenchmark uses its particular class feature in a single run. Its last column provides a normalization of the average running time to the original class system, which gives a percentage for the performance penalty in running time. Overall, many class and object operations have negligible overhead, and external field access is faster in the new class system due to optimizations uncovered by our refactoring of classes. Internal method calls have an additional
overhead of 7% because internal method lookup requires one level of indirection for vector indexing. The worst overhead is seen in subclassing operations, where the performance penalty can be as much as 11%. This increase in execution time is due to the need to maintain all of the vectors associated with the field and method tables. However, we expect that subclassing happens much less frequently than other class operations and certainly not in inner loops. In short, we do not consider the additional overhead detrimental to the workings of the class system.

We measure space efficiency by examining the space needed to represent classes and how that changes when adding fields or methods. We measure the space needed by a direct subclass of `object` with no fields or methods and direct subclasses of `object` that add some number of either fields or methods.

Because Racket does not provide exact memory accounting, our benchmark first queries the garbage collector for an estimate of memory use, creates a number of similar class values, and then asks for a new estimate. Dividing the difference in the estimates by the number of values created yields an estimate of the space needed for each class. We run each benchmark ten times, but in practice the variance for each run is much less than a single byte.

Figure 4.16 contains a graph of the resulting measurements. Each point represents the additional space required by the new class library. Our data shows that this increase in space is generally constant per additional feature, whether method or field. The only place where this increase is not constant is the change from a class with no methods to one that

<table>
<thead>
<tr>
<th>Figure 4.15 Micro-benchmark results (times in milliseconds)</th>
<th>rev</th>
<th>reps</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>External field access port</td>
<td>2.5 * 10⁶</td>
<td>1928</td>
<td>2016.88</td>
<td>2140</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>External field access class/c</td>
<td>2.5 * 10⁷</td>
<td>1696</td>
<td>1897.16</td>
<td>2248</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>Internal (same class) field access port</td>
<td>2.5 * 10⁸</td>
<td>3588</td>
<td>3665.84</td>
<td>4176</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Internal (same class) field access class/c</td>
<td>2.5 * 10⁸</td>
<td>3584</td>
<td>3632.12</td>
<td>3692</td>
<td>0.99</td>
<td></td>
</tr>
<tr>
<td>Internal (super class) field access port</td>
<td>2.5 * 10⁸</td>
<td>3588</td>
<td>3665.60</td>
<td>4184</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Internal (super class) field access class/c</td>
<td>2.5 * 10⁸</td>
<td>3584</td>
<td>3636.44</td>
<td>4252</td>
<td>0.99</td>
<td></td>
</tr>
<tr>
<td>External method access port</td>
<td>2.5 * 10⁷</td>
<td>1948</td>
<td>2076.28</td>
<td>3060</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>External method access class/c</td>
<td>2.5 * 10⁷</td>
<td>1964</td>
<td>2074.20</td>
<td>2196</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>Internal (same class) method access port</td>
<td>2.5 * 10⁷</td>
<td>852</td>
<td>889.72</td>
<td>936</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Internal (same class) method access class/c</td>
<td>2.5 * 10⁷</td>
<td>916</td>
<td>950.56</td>
<td>1000</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td>Internal (super class) method access port</td>
<td>2.5 * 10⁷</td>
<td>856</td>
<td>890.12</td>
<td>956</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Internal (super class) method access class/c</td>
<td>2.5 * 10⁷</td>
<td>924</td>
<td>952.40</td>
<td>988</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td>Object creation port</td>
<td>2.5 * 10⁷</td>
<td>5229</td>
<td>5457.52</td>
<td>6461</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Object creation class/c</td>
<td>2.5 * 10⁷</td>
<td>5317</td>
<td>5488.08</td>
<td>7377</td>
<td>1.01</td>
<td></td>
</tr>
<tr>
<td>Subclassing (no methods/fields) port</td>
<td>2.5 * 10⁷</td>
<td>3832</td>
<td>4020.08</td>
<td>4328</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Subclassing (no methods/fields) class/c</td>
<td>2.5 * 10⁷</td>
<td>3916</td>
<td>4077.16</td>
<td>4284</td>
<td>1.01</td>
<td></td>
</tr>
<tr>
<td>Subclassing (no fields) port</td>
<td>2.5 * 10⁷</td>
<td>4176</td>
<td>4377.04</td>
<td>4685</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Subclassing (no fields) class/c</td>
<td>2.5 * 10⁷</td>
<td>4493</td>
<td>4650.04</td>
<td>4905</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>Subclassing (no methods) port</td>
<td>2.5 * 10⁷</td>
<td>4040</td>
<td>4230.04</td>
<td>4604</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Subclassing (no methods) class/c</td>
<td>2.5 * 10⁷</td>
<td>4484</td>
<td>4660.04</td>
<td>4976</td>
<td>1.10</td>
<td></td>
</tr>
<tr>
<td>Subclassing (methods/fields) port</td>
<td>2.5 * 10⁷</td>
<td>4449</td>
<td>4644.40</td>
<td>5052</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Subclassing (methods/fields) class/c</td>
<td>2.5 * 10⁷</td>
<td>4948</td>
<td>5141.12</td>
<td>5444</td>
<td>1.11</td>
<td></td>
</tr>
</tbody>
</table>
4.5. EVALUATING CLASS CONTRACTS

Figure 4.16 Space benchmarks for classes

Difference in space benchmarks for pre- and post-contracts

| Fields | + |
| Methods | × |

contains methods. The large bump for the first added method represents parts of the class representation allocated when a given class contains methods.

4.5.2 Evaluating Class Contract Features

At first glance, our contract language for first-class classes is large. It comes with many more contract forms than the contract languages for conventional object-oriented programming languages. To evaluate the usefulness of our contract language, we add simple, type-like\(^5\) contracts to both Racket’s base GUI library and an extended GUI framework. These contracts are derived from the documentation for both libraries. In the GUI library, these contracts replace existing dynamic type checks. In contrast, the GUI framework had no dynamic checks to begin with. Our changes add about 2700 non-whitespace lines of contract code to the GUI library and 2000 lines to the framework library, which contain 30,000 and 20,000 lines of code respectively. About 460 lines that implement dynamic type checks were removed.

Figure 4.17 provides a breakdown of what kind of contracts we supply. It ignores all contracts for non-OO features. The majority of these contracts protect classes and describe

\(^5\)A type-like contract uses type predicates such as `number?` to check arguments, results, and fields.
CHAPTER 4. CONTRACTS FOR FIRST-CLASS CLASSES

**Figure 4.17 Classes, Interfaces, and Mixin Counts**

<table>
<thead>
<tr>
<th>Category</th>
<th>GUI</th>
<th>Framework</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>78</td>
<td>55</td>
<td>133</td>
</tr>
<tr>
<td>Interface</td>
<td>18</td>
<td>52</td>
<td>70</td>
</tr>
<tr>
<td>Mixin</td>
<td>0</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>All</td>
<td>96</td>
<td>154</td>
<td>250</td>
</tr>
</tbody>
</table>

how a client should interact with instances and how a subclass should specialize behavior. Others apply to interfaces and mixins.

As seen in our presentation of the Racket class system, mixins in Racket are just functions that map a class to a different class. Here is a typical mixin contract from our revised code base:

```racket
(define text:line-numbers-mixin/c
  (→ (and/c (subclass?/c text%) ...
        (class/c (augment after-insert ...)
                  (inherit [last-line (m → natural?)] ...))))
  ;; a contract for the text:line-numbers(%) interface:
  (and/c text%/c
    (class/c [show-line-numbers! (m → boolean? void?)]
     [show-line-numbers? (m → boolean?)]
     [set-line-numbers-color (m → string? void?)])))
```

The domain contract combines two class contracts via `and/c`. The first part ensures that the argument class is a subclass of the `text%` class. The second part uses `augment` and `inherit` to ensure that the given class defines two methods: `after-insert` and `last-line`; the former must be augmentable and the latter must return a natural number when called in the result class. The range contract ensures that the result satisfies the `text%/c` contract and implements three contracted methods. The `text%/c` contract adds method contracts to protect instances of the result class with the same method contracts as an ordinary text editor. The method contracts in the `class/c` form protect instances of the mixin’s result; for example, if a client program creates an instance of the result class and calls the `set-line-numbers-color` method, then this method expects a string as an argument.

This example shows the typical obligations that a mixin contract imposes on its argument classes and on its result classes: methods that must exist in the argument class and allowed uses of the result class by client code. Examples of the former include `inherit` and `super` clauses while examples of the latter include ordinary method contracts, `override`, and `inner` clauses.
While our contract system does not directly support the specification of behavioral subtyping on interfaces, it can approximate such relationships. In our GUI hierarchy, the `area(%)` interface is at the root, with `window(%)` immediately below. The expectation is that windows behave like graphical areas and support additional functionality. The following contracts check this expectation:

\[
\begin{align*}
(\text{define } \text{area}(\%)/c \\
(\text{class/c } [\text{get-parent (or/c (is-a?/c area-container\%)) false/c}]) \\
[\text{get-top-level-window (or/c (is-a?/c frame\%) (is-a?/c dialog\%))}]) \ldots))
\end{align*}
\]

\[
(\text{define } \text{window}(\%)/c \\
(\text{and/c } \text{area}(\%)/c \\
(\text{class/c } [\text{has-focus? (m \rightarrow boolean\%)}) \\
[\text{on-focus (m \rightarrow any/c void\%)}) \\
(\text{override [on-focus (m \rightarrow any/c void\%)}) \ldots))])
\end{align*}
\]

The `window(%)` contract uses the `and/c` combinator to add new method clauses that check the methods not already included in `area(%)`. Also, the `on-focus` method is included a second time in an `override` clause with the same contract. This idiom—where an event handling method is supposed to be overridden in subclasses and where constraints are specified in the base class—is commonly used in the design of the GUI library and is the main motivation for `override` contracts in our system. When an interface adds no additional methods of its own, we write contracts that are just the conjunction of two or more contracts, e.g.:

\[
(\text{define } \text{subwindow}(\%)/c (\text{and/c } \text{subarea}(\%)/c \text{window}(\%)/c))
\]

Figure 4.18 contains the number of times we use each feature of the contract system. The table confirms that the expressiveness of our system is needed, but unsurprisingly, ordinary method contracts constitute the vast majority of contract clauses.

<table>
<thead>
<tr>
<th>Feature</th>
<th>GUI</th>
<th>Framework</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>1197</td>
<td>462</td>
<td>1659</td>
</tr>
<tr>
<td>init</td>
<td>390</td>
<td>0</td>
<td>390</td>
</tr>
<tr>
<td>absent</td>
<td>0</td>
<td>205</td>
<td>205</td>
</tr>
<tr>
<td>override</td>
<td>135</td>
<td>147</td>
<td>282</td>
</tr>
<tr>
<td>augment</td>
<td>0</td>
<td>59</td>
<td>59</td>
</tr>
<tr>
<td>inner</td>
<td>53</td>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>inherit</td>
<td>0</td>
<td>292</td>
<td>292</td>
</tr>
<tr>
<td>super</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Another insight concerns the use of specialization interfaces [Lamping, 1993], especially in the GUI libraries. Recall that Racket supports Beta-style inheritance. Methods can be
specialized by either overriding or augmenting at a particular point in the class hierarchy. A single class cannot have a subclass that augments a given method and another subclass that overrides it. Thus, whether a method may be overridden or augmented is an important part of the specification interface. To check such specifications, we add contract clauses that, like the documentation, specify which extension mechanism is in effect for a given method. This explains why we see many uses of inner and override contracts in the two GUI libraries, which heavily rely on both kinds of client-defined refinements.

Adding class contracts to our code base exposed several inconsistencies between the documentation and the implementation of the GUI libraries. For example, a method that should have returned menu widgets—and was used by existing client code with that expectation—was documented as returning void instead.

The contracts also uncovered bugs in mixins. In one particular case, the documentation described a mixin as adding a particular interface declaration to the result class. Instead, the mixin added a different, incorrect interface. Since the added contracts checked that both the argument and result classes in a mixin implement the correct interfaces, running the IDE tests caught this error.

Finally, our contracts found cases where behavioral subtyping is intended yet violated. In one case, a class in a test driver was subclassed from an editor class that defined override contracts for its methods. The superclass defined a method with optional arguments, but the test driver’s class overrode this method with only mandatory arguments. This change, in turn, broke the expectation of client code, which expected a method specification with optional arguments.

4.5.3 Performance Evaluation

To evaluate the performance of the contract system, we use the DrRacket test suite—a comprehensive set of tests that simulates interactions with the IDE. The test suite utilizes both the base GUI library and the extended GUI framework. It sends simulated keyboard and mouse events to the DrRacket UI.

<table>
<thead>
<tr>
<th>run</th>
<th>max</th>
<th>mean</th>
<th>min</th>
<th>norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Checks or Contracts</td>
<td>747847</td>
<td>728874</td>
<td>722926</td>
<td>0.97</td>
</tr>
<tr>
<td>No Contracts</td>
<td>788179</td>
<td>749939</td>
<td>741486</td>
<td>1.00</td>
</tr>
<tr>
<td>Only GUI Contracts</td>
<td>809095</td>
<td>796642</td>
<td>792165</td>
<td>1.06</td>
</tr>
<tr>
<td>GUI and Framework Contracts</td>
<td>900266</td>
<td>821175</td>
<td>807671</td>
<td>1.09</td>
</tr>
</tbody>
</table>

It also revealed the need for the absent clause, discussed in section 4.1.1.
Because the original GUI library uses dynamic checks in lieu of contracts, we also measure the amount of overhead introduced by these original checks. Figure 4.19 contains the results from running the test suite with no dynamic checks or contracts, with just dynamic checks, with contracts added to the GUI library, and with contracts added to both the GUI and framework libraries. The tests were executed ten times on an AMD Phenom II X2 with 4GB of RAM running Debian GNU/Linux. The results show that the contracts on the GUI library add around 6% overhead compared to the original dynamic checks. Because the added contracts are stronger than the original dynamic checks, part of this additional overhead is due to the strengthened checking. The contracts on the framework library—which had no dynamic checking built-in to begin with—add an additional 3% overhead. Since the Racket IDE is a highly interactive program, we consider this runtime cost acceptable in exchange for more precise checks and better error reporting.

To provide a more fine-grained account for the overhead, figure 4.20 reports the number of times that particular portions of the method contracts were executed. The flat checks guarantee properties that can be checked immediately when applying the contract to the class. As with with normal function contracts, the domain and range checks are performed at each method call site. Note that the execution counts in the figure are scaled differently for each component. As expected, method domain and range checks occur much more frequently than flat checks because these checks occur at every invocation. An unusual aspect of the distribution is that range checks appear much more frequently than domain checks. A code inspection suggests that this imbalance is due to the frequent use of nullary methods in object-oriented style code.
4.6 Interface Contracts

Now that we have shown how to protect particular instances of classes, we look at how to protect sets of classes via contracted interfaces. Racket’s class system contains interfaces like those found in Java. That is, an interface in Racket is a (possibly empty) set of superinterfaces and a collection of method names that implementing classes must contain:

\[
\begin{align*}
&\text{(define animal(%) (interface () eat full))} \\
&\text{(define land-animal(%) (interface (animal(%) walk run))}
\end{align*}
\]

Here, the interface animal(%) has no superinterfaces and it requires implementing classes to contain the public methods eat and full?. The interface land-animal(%) has animal(%) as a superinterface, so it still requires eat and full?, and it adds two new required methods, walk and run.

Classes that implement interfaces are defined via the class\(^*\) form. This form differs from the class form in that it adds a collection of interfaces that the class claims to implement. Here we add an implemented interface to the fish% class from section 4.1:

\[
\begin{align*}
&\text{(define fish%} \\
&\quad \text{(class* object% (animal(%)}) \\
&\quad \quad \text{(super-new}) \\
&\quad \quad \text{(init-field name}) \\
&\quad \quad \text{(field [weight 10])} \\
&\quad \quad \text{(define max-weight 20)} \\
&\quad \quad \text{(define/public (full?)} \\
&\quad \quad \quad (>= weight max-weight)) \\
&\quad \quad \text{(define/public (eat f)} \\
&\quad \quad \quad (set! weight (+ weight (get-field weight f))))
\end{align*}
\]

If we elided the full? method, for example, then we would get an error at class creation time listing the missing methods.

4.6.1 Contracts on Interfaces

To add contracts to interfaces, we extend the syntax for interfaces to pair a method name with a contract. Here is an illustrative example involving an extended version of the animal(%) interface:
We use the contracts for \texttt{eat} and \texttt{full?} from \texttt{fish/c} in section 4.1.1. The new method, \texttt{spawn}, returns a new animal when invoked.

We can refine contracts in subinterfaces:

\begin{verbatim}
(define insect(\%)
  (interface (animal(\%)) [spawn (m! insect(\%))]))
(define mammal(\%)
  (interface (animal(\%)) [spawn (m! mammal(\%))]))
\end{verbatim}

Here, a class that implements one of these subinterfaces should return an object from \texttt{spawn} that matches the same subinterface.

However, not all refinements are appropriate:

\begin{verbatim}
(define herbivore<\%>
  (interface (animal(\%))
    [eat (m! plant(\%) #:pre (not (send this full?)) void?)])
(define carnivore(\%)
  (interface (animal(\%))
    [eat (m! animal(\%) #:pre (not (send this full?)) void?)])
\end{verbatim}

At first glance, both \texttt{herbivore<\%>} and \texttt{carnivore(\%)} seem useful. However, a \texttt{herbivore<\%>} (likewise, \texttt{carnivore(\%)}) cannot be used in the same contexts as a \texttt{animal(\%)}, because a \texttt{herbivore<\%>} eats only plants (likewise, animals) while an \texttt{animal(\%)} eats any type of food. That is, a client that only knows that a given object is an \texttt{animal(\%)} might try to feed it an \texttt{animal(\%)}, but if that object is a \texttt{herbivore<\%>}, then the client will have unwittingly broken the contract. This means that the two subinterfaces fail to be behavioral subtypes [Liskov and Wing, 1994] of their superinterface. Instead of blaming the client for breaking the inappropriately restricted contract, the subinterface should instead be blamed for failing to be a subtype of the superinterface [Findler and Felleisen, 2001].

To ensure a subinterface is a behavioral subtype of its superinterface, the contracts added by the subinterface should imply any corresponding contracts that are in the superinterface. This means that positive positions of the new contracts should imply the corresponding positions in the superinterface’s contracts, and negative positions should be implied by the corresponding positions of the superinterface’s contract. That is, positive positions may
involve stronger checks than in the superinterface’s contract, whereas negative positions are restricted to weaker checks.

### 4.6.2 Converting Interface Contracts to Projections

Due to behavioral subtyping, we can have three different sources of blame: the class that implements a set of interfaces, the client of that class, and any interfaces whose contracts fail to imply their superinterface’s contracts. In order to apply contracts to values, we need to pick the appropriate positive and negative blame for each contract. In addition, the order in which we apply the projections becomes important so that we get behavioral subtyping checking while applying each projection only once.

To illustrate how to convert a series of interface contracts into an applicable list of projections, we use the following example program:

```scheme
(define i1 (interface () [m c1] ...))
(define i2 (interface (i1) [m c2] ...))
(define i3 (interface (i1) ...))
(define i4 (interface (i2 i3) [m c4] ...))
(define i5 (interface (i3) [m c5] ...))
(define i6 (interface () ...))
(define i7 (interface (i6) [m c7] ...))
(define i8 (interface (i6) ...))
(define i9 (interface (i7 i8) ...))
(define cls (class* object% (i4 i5 i9) ...))
(define usr (new cls))
```

Here, `usr` plays the role of the client. We provide a pictorial representation of this interface hierarchy in figure 4.21. Here, solid squares represent interfaces that include a contract for `m` and dashed squares those without.

We choose blame parties for each contract as follows. The first two rules apportion blame for the class and its client, and the latter two apportion blame for breaking behavioral subtyping.

- The contracts closest to the leaves of the hierarchy have the class `cls` as positive blame.
- The contracts closest to the roots of the hierarchy have the client `usr` as negative blame.
• The contracts from superinterfaces that have at least one subinterface with a contract are given one of those subinterfaces as positive blame.

• The contracts from subinterfaces that have at least one superinterface with a contract are given the subinterface itself as the negative blame.

The last rule ensures that the subinterface, not the superinterface, is blamed for not being a behavioral subtype if the superinterface’s contract succeeds in a negative position but the subinterface’s fails.

In order to ensure that this blame apportionment appropriately checks the implications involved in behavioral subtyping, we must apply the resulting projections from subinterfaces before those for superinterfaces. This ensures the positive positions of subinterface contracts are checked before the positive position of superinterface contracts, and vice versa for the negative positions.

With these rules in mind, here is a correct projection composition for the hierarchy in figure 4.21:

\[ c_{i_2,usr} \circ c_{i_4,i_2} \circ c_{cls,i_4} \circ c_{cls,i_5} \circ c_{cls,usr} \]

Here, each contract is superscripted with its positive and negative blame parties, in that order. The only ordering restrictions are that contract \( c_4 \) is checked before both \( c_2 \) and \( c_1 \), \( c_5 \) is checked before \( c_1 \), and \( c_2 \) is checked before \( c_1 \). Thus, the following is another possible projection ordering:

\[ c_{cls,usr} \circ c_{i_2,usr} \circ c_{i_4,i_2} \circ c_{cls,i_5} \circ c_{cls,i_4} \]

This final example of a correct projection composition for figure 4.21 shows that the choice of positive blame for superinterfaces with multiple subinterfaces is not unique:
Here, \( i_5 \) serves as the positive blame for \( i_1 \)’s contract \( c_1 \) instead of \( i_2 \). This is also a correct blame apportionment because both \( c_5 \) and \( c_2 \) have been checked when \( c_1 \) is checked, and if both succeeded, then neither implies \( c_1 \) if \( c_1 \) fails.

### 4.6.3 Implementing Interface Contracts

Generating the appropriate projections is not the whole story, however. Until a class containing interface contracts is instantiated, the contract system does not know the blame party that acts as the class client. Therefore, we must store enough information in the class representation to appropriately generate the necessary projections at object creation.

When a new class is created, the class system traverses the interface hierarchy and collects all the contracts as well as the appropriate blame parties for all but the uppermost contracts in the hierarchy—that is, those that have the class client as negative blame. In addition, the contracts are appropriately ordered as described in the previous section for later application. All this information is stored in the method table along with the method implementation, and the names of the methods that contain this information instead of just a method implementation are also stored in the class.

During object creation, the list of interface contracted methods is checked. If it is empty, then the class is instantiated directly. Otherwise, a new class is generated by using the blame information for the region containing the instantiation as the missing negative blame. The newly generated class contains appropriately protected methods in the method table in the place of the old interface contract information. This new class is then instantiated to create the requested object.

To avoid generating a new class every time a class with interface contracts is instantiated, the class system also stores a weak hash table in the class representation that maps blame to contracted classes. Thus, the first time an interface contracted class is instantiated in a given blame region, a new class is appropriately generated. On subsequent instantiations, the already generated class is retrieved from the hash table.

Figure 4.22 contains some of the same microbenchmarks that were used to test the impact of the changes to the class system for class contracts. We drop the benchmarks for field and method access, since the changes to add interface contracts do not affect these operations at all. As before, each benchmark is run one hundred times, and we report the average, minimum, and maximum running times. The “rev” column contains “plain” for the class system prior to adding interface contracts and “ifc” for the new class system.

Object creation shows an additional 7% overhead, but this is due to the fact that our microbenchmark uses classes that do not contain any instantiation-time computation. Thus,
4.6. INTERFACE CONTRACTS

Figure 4.22 Micro-benchmark results (times in milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>rev</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object creation</td>
<td>plain</td>
<td>5369</td>
<td>5633.00</td>
<td>6185</td>
<td>1</td>
</tr>
<tr>
<td>Object creation</td>
<td>ifc</td>
<td>5789</td>
<td>6011.76</td>
<td>6629</td>
<td>1.07</td>
</tr>
<tr>
<td>Subclassing (no methods/fields)</td>
<td>plain</td>
<td>4392</td>
<td>4559.59</td>
<td>5073</td>
<td>1</td>
</tr>
<tr>
<td>Subclassing (no methods/fields)</td>
<td>ifc</td>
<td>5024</td>
<td>5244.07</td>
<td>5693</td>
<td>1.15</td>
</tr>
<tr>
<td>Subclassing (no fields)</td>
<td>plain</td>
<td>4928</td>
<td>5111.32</td>
<td>5748</td>
<td>1</td>
</tr>
<tr>
<td>Subclassing (no fields)</td>
<td>ifc</td>
<td>5712</td>
<td>5876.76</td>
<td>6332</td>
<td>1.15</td>
</tr>
<tr>
<td>Subclassing (no methods)</td>
<td>plain</td>
<td>4917</td>
<td>5115.72</td>
<td>5696</td>
<td>1</td>
</tr>
<tr>
<td>Subclassing (no methods)</td>
<td>ifc</td>
<td>5745</td>
<td>5963.24</td>
<td>6504</td>
<td>1.17</td>
</tr>
<tr>
<td>Subclassing (methods/fields)</td>
<td>plain</td>
<td>5444</td>
<td>5624.49</td>
<td>6261</td>
<td>1</td>
</tr>
<tr>
<td>Subclassing (methods/fields)</td>
<td>ifc</td>
<td>6313</td>
<td>6514.68</td>
<td>7017</td>
<td>1.16</td>
</tr>
</tbody>
</table>

Run on an Intel Core i7 860 (2.8GHz) with 8 GB of DDR3-1333 memory running Ubuntu 11.10.

the extra check for methods protected by interface contracts, even though there are none, adds a small but noticeable overhead. The 15% overhead in subclassing (the least frequently used class operation) is also magnified by the small classes used for the microbenchmarks. In return for this performance degradation, programmers receive improved information when contracts fail and the ability to equip interfaces with contracts.
Chapter 5

Contracts for Mutable Values\(^1\)

This chapter presents contracts for mutable values. It argues that proper support for these contracts requires changes to the operations for these values. We then describe two types of proxies, called chaperones and impersonators, that allow us to implement the desired contracts. We evaluate our design in two ways: we model proxies and show how they protect the desired invariants, and we compare our implementation of proxies in Racket against rival proxy systems.

5.1 Extensibility versus Reasoning

An extensible programming language such as 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 making it harder 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 runtime 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.

Here we describe an addition to the Racket core that enables language constructs to be implemented in a library where the constructs depend on run-time \textit{interposition}—or

\(^1\)This is joint work with Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt [Strickland et al., 2012].
**CHAPTER 5. CONTRACTS FOR MUTABLE VALUES**

*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*. Both chaperones and impersonators are proxies, i.e., a wrapper object that
interposes on operations intended for a target object. Chaperones can only constrain the
behaviors of the objects that they wrap; for an interposed operation, a chaperone must either
raise an exception or return the same value as the operation on the original object, possibly
wrapped with a chaperone. Impersonators, in contrast, are relatively free to change the
behavior of the objects that they wrap or return different values. In general, impersonators
are more expressive than chaperones, but the latter are allowed on more kinds of values.

Together, chaperones and impersonators are powerful enough to implement the desired
contract system without subverting guarantees that enable composition of language exten-
sions. Thanks to chaperones and impersonators, the Racket contract system now supports
higher-order contracts on mutable objects and generative structs. This improvement di-
rectly 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 pat-
terns, such as transparent access of remotely stored fields, while chaperones can be used to
implement constructs such as revokable membranes [Miller, 2006].

Last but not least, building just interposition support into the core compiler and run-
time system offers the promise of improved performance, both for code that uses libraries
such as contracts and code that does not. Core-language support for interposition has
already eliminated a factor of three slowdown for some object-oriented operations in pro-
grams that did not use contracts at all. Contract support for class-based objects now
affects performance only for programs that use contracts. 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 invariants.
Section 5.2 uses contract checking in Racket to explore issues of expressiveness and invariants related to interposition. Section 5.3 describes Racket’s chaperone and impersonator API and relates it to the implementation of contracts. Section 5.4 presents a formal model for a subset of Racket with chaperones and impersonators and proves our invariants; the full model and proofs appear in appendix B. Section 5.5 reports performance numbers.

5.2 Interposition via Contracts

Contract checking is easily the most prominent use of interposition in Racket. Furthermore, 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 an exploration of Racket contracts as a way of motivating the design of chaperones and impersonators.

5.2.1 Predicates and Function Contracts

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

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

\begin{verbatim}
(math.rkt)
(define pi (* (acos 0) 2))
(provide/contract [pi real?])

(circles.rkt)
(require "math.rkt")
pi
\end{verbatim}

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

Not all contract checks can be performed immediately when a value crosses a boundary. Some contracts require a delayed check [Findler and Felleisen, 2002], which is like a boundary wrapped around a value:
Delayed checks are needed for function contracts, such as when `math.rkt` exports a `sqr` function that is used by `circles.rkt`:

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

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

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

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

```
(deriv cos) -> (real? . -> .
     nonnegative-real?))]])
```

When a function consumes another function as an argument, then a check-delaying wrapper on the consuming function can add delayed checks to the argument function. For example, a numerically approximating `derivative` consumes a function that must accept real numbers and return real numbers:
Along the same lines, a function may capture a wrapped function in its closure. The result produced by \texttt{derivative}, for example, closes over the argument with the argument’s wrapper intact:

\begin{center}
\begin{tikzpicture}
    
    \node[draw, circle, fill=gray!50] (a) at (0,0) {};
    \node[draw, circle, fill=gray!50] (b) at (1,0) {};
    \draw[->] (a) -- (b);
\end{tikzpicture}
\end{center}

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

\begin{center}
\begin{Verbatim}
\begin{tabular}{|l|}
\hline
\texttt{poly.rkt} & \texttt{client.rkt} \\
\hline
\texttt{(define (id x) x)} & \texttt{(require "poly.rkt")} \\
\texttt{(provide/contract} & \texttt{(id 199.99)} \\
\texttt{  [id (let ([\alpha (new-\forall/c)])} & \texttt{)} \\
\texttt{    (\alpha . -> . \alpha))]} & \texttt{)} \\
\hline
\end{tabular}
\end{Verbatim}
\end{center}

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

\begin{center}
\begin{tikzpicture}
    
    \node[draw, circle, fill=gray!50] (a) at (0,0) {};
    \node[draw, circle, fill=gray!50] (b) at (1,0) {};
    \draw[->] (a) -- (b);
\end{tikzpicture}
\end{center}

When \texttt{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:\footnote{Matthews and Ahmed [2008] show that this wrapper protocol implements parametricity.}

\begin{center}
\begin{tikzpicture}
    
    \node[draw, circle, fill=gray!50] (a) at (0,0) {};
    \node[draw, circle, fill=gray!50] (b) at (1,0) {};
    \draw[->] (a) -- (b);
\end{tikzpicture}
\end{center}

As originally implemented for Racket, simple predicate contracts, function contracts, and even \texttt{new-\forall/c} require no particular run-time support; function wrappers are easily implemented with \texttt{\lambda} and opaque wrappers via Racket’s \texttt{struct} form. Run-time support becomes necessary, however, to generalize contracts beyond immediate predicates and wrappers for purely functional objects.
5.2.2 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:

```
(define constants
  (list 8.02e+23 6.6e-11))
(provide/contract
  [constants
    (listof nonnegative-real?)])
```

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:

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

```
5.2. INTERPOSITION VIA CONTRACTS

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:

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 need to add a wrapper:

Finally, installing a function into the vector may also require 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 it 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.

### 5.2.3 Structure Contracts

Besides functions and built-in datatypes like lists and vectors, Racket provides constructs to define new structure types. Reliable structure-type opacity is crucial in the Racket ecosystem. Not only ordinary user libraries must 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:

```racket
(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 nearby `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 argument must be either `#f` or itself a widget, otherwise the construction is rejected.

```racket
(define (widget-root w)
  (let ((p (widget-parent w)))
    (if p (widget-root p) null))

(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.

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

```racket
(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 `scene.rkt`
module below promises that plot is an instance of the widget struct whose first field is an OpenGL window.

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

circles.rkt
(replace "widget.rkt"
 "scene.rkt")

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.

### 5.2.4 Parameteric 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
(replace "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. Specifically, it gets a value for `plot`’s parent that is neither `#f` or a `widget`, despite the `widget #:guard` and the `widget?` argument contract.

Therefore, allowing `(struct/c widget (new-∀/c) ....)` would be broken. 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` could have wrapped the procedure to ensure properties of the function.

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.

5.2.5 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 wrappers.

- **Higher-order contracts** require wrappers to delay checks.

In generalizing contracts to compound data types and programmer-defined structure, 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 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 \( \rightarrow \) 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 widgets, because 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:

<table>
<thead>
<tr>
<th>flat contracts</th>
<th>higher-order contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedures</td>
<td>✓</td>
</tr>
<tr>
<td>immutable data</td>
<td>✓</td>
</tr>
<tr>
<td>mutable data</td>
<td>✓</td>
</tr>
<tr>
<td>immutable fields</td>
<td>✓</td>
</tr>
<tr>
<td>mutable fields</td>
<td>✓</td>
</tr>
<tr>
<td>objects</td>
<td>✓*</td>
</tr>
</tbody>
</table>

The tildes indicate points where flat contracts were allowed for mutable data and mutable fields of structures. In these cases, the contracts were checked only partly, because mutation could subvert the checks. 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:

<table>
<thead>
<tr>
<th>flat contracts</th>
<th>chaperone contracts</th>
<th>impersonator contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedures</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>immutable data</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mutable data</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>immutable fields</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mutable fields</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>objects</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The unsupported cases in this table are 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.
5.3 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 plus additional proxy patterns.

Figure 5.1 shows part of the Racket chaperone and impersonator API. 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.

5.3.1 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 replace the arguments to original function or the results from the original function.

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

\[(\text{chaperone-procedure} \text{ orig-proc} \text{ interpose-proc})\]

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

\[(\lambda \text{ (a b)} \ldots \text{ (values new-a new-b)})\]

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

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 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.

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.

---

3The complete API is about twice as large [Flatt and PLT, 2010, §13.5].
5.3. CHAPERONES AND IMPERSONATORS

Figure 5.1 Partial chaperone and impersonator API

(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.

...

Since the post-interpose-proc is determined after the arguments are available, the replacement result from post-interpose-proc may 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

\[
\lambda (a \ b) \ldots \ (values \ new-a \ new-b
   \ ; \ post-interpose-proc:
      (\lambda (result) \ldots \ new-result)))
\]

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

Using chaperones to implement contracts is straightforward. The contract on a procedure like \(\text{sqr}\),

\[
\text{(provide/contract}
   \ [\text{sqr} \ (\text{real?} \ . \ \rightarrow \ . \ \text{nonnegative-real?})])
\]

is implemented as a chaperone of \(\text{sqr}\):

(chaperone-procedure
   \text{sqr}
   (\lambda (n)
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 \(\text{\texttt{\_\_}}\), 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.

### 5.3.2 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 of a vector 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:

\[
\text{chaperone-vector vec interpose-ref interpose-set}
\]

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, this contract for partial-sums!:

\[
\text{(provide/contract}
\text{[partial-sums! ((vectorof number?) \_\_ \_\_ \_\_any\_\_\_\_\_)])}
\]
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:

```scheme
(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`, 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.
5.3.3 Chaperoning and Impersonating Structures

As noted in section 5.2.3, 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 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 together 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 functions such as 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. For similar reasons as impersonate-vector, impersonate-struct only allows interposition on mutable fields of a structure.

5.4 Reasoning about Reasonable Interposition

This section presents a formal model of chaperones and impersonators. Using the model, we can formally establish limits on the power of interposition and state a precise theorem.

---

4In practice, most (but not all) procedures and vectors have specialized representations that are exploited by the just-in-time compiler.
In this section, we explain the kinds of invariants that chaperones are meant to preserve. To set the stage for our formal model preservation of invariants, we define VectorRacket, which is a subset of Racket with both mutable and immutable vectors. We equip VectorRacket with chaperones and impersonators for vectors, and we then present our proxy theorem for chaperones.

5.4.1 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 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 is no different, however, 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 already constrained 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, which 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 our attention to procedures and mutable and immutable vectors, since the structure-type generativity can be simulated through vectors, procedures, and hidden type tags.
5.4.2 VectorRacket

Figure 5.2 shows the grammar for VectorRacket. The surface language (the left-hand column) includes \( \lambda \) 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 in figure 5.3 returns the atomic tag proc or vector to indicate that the result of evaluation was some procedure or 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 the evaluation of a program fails to terminate.

The evaluator is specified via the reduction relation \( \rightarrow \), which is shown in figure 5.4. The relation uses the additional syntactic categories given on the right-hand column of figure 5.2. 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 language for expressions is nearly the same as the set of surface-level expressions, with the exception that the production \((\text{loc } x)\) is added.
5.4. REASONING ABOUT REASONABLE INTERPOSITION

5.4.1 Evaluation Contexts

and 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 \texttt{[procedure]} rule, so the \texttt{[\beta v]} rule extracts the procedure from the store before substitution. The rules for \texttt{if} treat non-\texttt{#f} values as if they were true (as in Racket).

The \texttt{[equal?]} rule defers to the \texttt{equal} metafunction is given in figure 5.7. It returns \texttt{#t} when the (potentially infinite) unfolding of the first argument is equal to the (potentially infinite) unfolding of the second. The \texttt{immutable?} predicate detects immutable vectors.

The remaining rules handle vector allocation, access, and update, where vector allocation records whether it was allocated by an interposition (i.e., the program state’s boolean).

5.4.3 VectorRacket with Chaperones

Figure 5.5 extends the syntax of VectorRacket with chaperones and impersonators. The extensions include three new primitives, value forms for chaperones and impersonators, and \texttt{set-marker} and \texttt{clear-marker} forms to record whether evaluation has entered a chaperone’s interposition functions.

The \texttt{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:

\[
\text{Eval} \left[ \text{vector-ref} \left( \text{chaperone-vector} \left( \text{vector} \ 1 \ 2 \ 3 \right) \right) \right. \\
\left. \left( \lambda \left( \text{vec} \ 1 \ \text{ov} \right) \ ov \right) \\
\left( \lambda \left( \text{vec} \ 1 \ \text{ov} \right) \ ov \right) \right] = 2 \\
1
\]

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

\[
\text{Eval} \left[ \text{vector-ref} \left( \text{chaperone-vector} \left( \text{vector} \ 1 \ 2 \ 3 \right) \right) \right. \\
\left. \left( \lambda \left( \text{vec} \ 1 \ \text{ov} \right) \ 17 \right) \\
\left( \lambda \left( \text{vec} \ 1 \ \text{ov} \right) \ ov \right) \right] = \text{error: bad-cvref}
\]

The \texttt{[out-cvec-ref]} and \texttt{[in-cvec-ref]} rules of figure 5.6 handle \texttt{vector-ref} for chaperones. The two rules are essentially the same, but \texttt{[out-cvec-ref]} applies when evaluation first moves into interposition mode, while \texttt{[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 \texttt{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
Figure 5.4 VectorRacket Reductions

\[
\begin{align*}
(s \ b \ E \[\lambda (y \ldots) \ e]) & \quad \text{[procedure]} \\
\rightarrow & \quad (s \ x \ e \[\lambda (y \ldots) \ e] \ b \ E \[\text{let} \ ((x \vdash v \ldots) \ e)]) & \quad \text{where } x \text{ fresh} \\
\rightarrow & \quad (s \ b \ E \[(x \vdash v \ldots) \ e]) & \quad \beta v \\
\rightarrow & \quad (s \ b \ E \[(\text{let} \ ((x \vdash v \ldots) \ e)) = s(x), (v \ldots) \vdash (x \ldots)]) \rightarrow \ b \ E \[\text{let} \ ((x \vdash v \ldots) \ e]) & \quad \text{[let]} \\
\rightarrow & \quad b \ E \[\text{if} \ (v \vdash e, e)] & \quad \text{[if]} \\
\rightarrow & \quad (s \ b \ E \[(\text{error} \ 'variable')]) & \quad \text{[error]} \\
\rightarrow & \quad (s \ b \ (\text{error} \ 'variable')) & \quad \text{where } [] \not\vdash E \\
\rightarrow & \quad (s \ b \ E \[(\text{equal?} \ v_1 v_2)]) & \quad \text{[equal?]} \\
\rightarrow & \quad (s \ b \ E \[(\text{equal?} \ s, v_1 v_2)]) & \quad \text{[vector]} \\
\rightarrow & \quad (s \ x \ e \[\text{vector} \ v \ldots] \ b \ E \[\text{let} \ ((x \vdash v \ldots) \ e)]) & \quad \text{where } x \text{ fresh} \\
\rightarrow & \quad (s \ b \ E \[(\text{immutable?} \ v)]) & \quad \text{[immut]} \\
\rightarrow & \quad (s \ b \ E \[(\text{immutable?} \ s, v)]) & \quad \text{[immvec]} \\
\rightarrow & \quad (s \ x \ e \[\text{vector-immutable} \ v \ldots] \ b \ E \[\text{let} \ ((x \vdash v \ldots) \ e)]) & \quad \text{where } x \text{ fresh} \\
\rightarrow & \quad (s \ x \ e \[\text{vector-print} \ (\text{loc} \ x) \ n \ v_0 \ldots]) & \quad \text{[vector-set!]} \\
\rightarrow & \quad (s \ x \ e \[\text{vector-print} \ v_1 \ldots v_0 \vdash \text{void}] \ b \ E \[\text{let} \ ((x \vdash v \ldots) = s(x), (v_1 \ldots) \vdash (x \ldots))] & \quad \text{[mvec-ref]} \\
\rightarrow & \quad (s \ b \ E \[v_2]) & \quad \text{where } (\text{vector-print} \ b_2 v_1 \ldots v_0 \vdash \text{void}) \ b \ E \[\text{let} \ ((x \vdash v \ldots) = s(x), (v_1 \ldots) \vdash (x \ldots))] & \quad \text{[immvec-ref]} \\
\rightarrow & \quad (s \ b \ E \[v_2]) & \quad \text{where } (\text{vector-immutable-print} \ b_2 v_1 \ldots v_0 \vdash \text{void}) \ b \ E \[\text{let} \ ((x \vdash v \ldots) = s(x), (v_1 \ldots) \vdash (x \ldots))] & \quad \text{[mvec-ref]} \\
\rightarrow & \quad (s \ b \ E \[v_2]) & \quad \text{where } (\text{vector-immutable-print} \ b_2 v_1 \ldots v_0 \vdash \text{void}) \ b \ E \[\text{let} \ ((x \vdash v \ldots) = s(x), (v_1 \ldots) \vdash (x \ldots))] & \quad \text{[immvec-ref]} \\
\rightarrow & \quad (s \ b \ E \[v_2]) & \quad \text{where } (\text{vector-immutable-print} \ b_2 v_1 \ldots v_0 \vdash \text{void}) \ b \ E \[\text{let} \ ((x \vdash v \ldots) = s(x), (v_1 \ldots) \vdash (x \ldots))] & \quad \text{[mvec-ref]} \\
\rightarrow & \quad (s \ b \ E \[v_2]) & \quad \text{where } (\text{vector-immutable-print} \ b_2 v_1 \ldots v_0 \vdash \text{void}) \ b \ E \[\text{let} \ ((x \vdash v \ldots) = s(x), (v_1 \ldots) \vdash (x \ldots))] & \quad \text{[immvec-ref]} \\
\end{align*}
\]

Figure 5.5 VectorRacket Chaperone Syntax Extensions

\[
\begin{align*}
\text{prim} :::= & \quad \ldots \ | \ \text{chaperone-vector} \ | \ \text{chaperone-of?} \ | \ \text{impersonate-vector} \\
\text{sv} :::= & \quad \ldots \ | \ \text{chaperone-vector} \ l \ m \ o \ | \ \text{impersonate-vector} \ l \ m \ o \\
\text{e} :::= & \quad \ldots \ | \ \text{set-marker} \ e \ | \ \text{clear-marker} \ e
\end{align*}
\]
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 [clear] helper rules directly manipulate the boolean in the program state and then reduce to their arguments.

Figure 5.6 VectorRacket Chaperone Reductions

\[
(s \#fE((vector-ref (loc x) n))) \quad [\text{out-cvec-ref}]
\]

\[
\rightarrow (s \#fE((let ((old (vector-ref l n)))
\hspace{1em} (let ((new (set-marker (m l n old))))
\hspace{1em} (clear-marker
\hspace{1em} (if (chaperone-of? new old)
\hspace{1em} new
\hspace{1em} (error 'bad-cvref)))))))
\hspace{1em} where (chaperone-vector l m o) = s(x)
\]

\[
(s \#tE((vector-ref (loc x) n))) \quad [\text{in-cvec-ref}]
\]

\[
\rightarrow (s \#tE((let ((old (vector-ref l n)))
\hspace{1em} (let ((new (m l n old))))
\hspace{1em} (if (chaperone-of? new old)
\hspace{1em} new
\hspace{1em} (error 'bad-cvref)))))))
\hspace{1em} where (chaperone-vector l m o) = s(x)
\]

\[
(s \#b E[(set-marker e)]) \rightarrow (s \#t E[e]) \quad [\text{setm}]
\]

\[
(s \#b E[(clear-marker e)]) \rightarrow (s \#f E[e]) \quad [\text{clear}]
\]

\[
(s \#fE((vector-set! (loc x) n v))) \quad [\text{out-cvec-set!}]
\]

\[
\rightarrow (s \#fE((let ((new (set-marker (o l n v))))
\hspace{1em} (clear-marker
\hspace{1em} (if (chaperone-of? new v)
\hspace{1em} (vector-set! l n new)
\hspace{1em} (error 'bad-cvset)))))))
\hspace{1em} where (chaperone-vector l m o) = s(x)
\]

\[
(s \#tE((vector-set! (loc x) n v))) \quad [\text{in-cvec-set!}]
\]

\[
\rightarrow (s \#tE((let ((new (o l n v))))
\hspace{1em} (if (chaperone-of? new v)
\hspace{1em} (vector-set! l n new)
\hspace{1em} (error 'bad-cvset)))))))
\hspace{1em} where (chaperone-vector l m o) = s(x)
\]

\[
(s \#b E[(vector-ref (loc x) n)]) \quad [\text{ivec-ref}]
\]

\[
\rightarrow (s \#b E((vector-ref l n)))
\hspace{1em} where (impersonate-vector l m o) = s(x)
\]

\[
(s \#b E[(vector-set! (loc x) n v)]) \quad [\text{ivec-set!}]
\]

\[
\rightarrow (s \#b E[(vector-set! l n (o l n v))])
\hspace{1em} where (impersonate-vector l m o) = s(x)
\]

\[
(s \#b E[(chaperone-vector l m o)]) \quad [\text{cvec}]
\]

\[
\rightarrow (s[x->(chaperone-vector l m o)] b E[(loc x)])
\hspace{1em} where isvector[s, l], x fresh
\]

\[
(s \#b E[(impersonate-vector l m o)]) \quad [\text{ivec}]
\]

\[
\rightarrow (s[x->(impersonate-vector l m o)] b E[(loc x)])
\hspace{1em} where isimmutable[s, l], x fresh
\]

\[
(s \#b E[(chaperone-of? v, v)]) \quad [\text{cof}]
\]

\[
\rightarrow (s \#b E[(chaperone-of?[s, v, v])])))
\]

The chaperone-of? primitive defers to the chaperone-of metafunction of figure 5.7. 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.

**Figure 5.7** VectorRacket Chaperone Metafunctions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>immutable![[s, (loc x)]] = #t</code></td>
<td>where (vector-immutable v ...) = s(x)</td>
</tr>
<tr>
<td><code>immutable![[s, (loc x)]] = immutable![[s, l]]</code></td>
<td>where (chaperone-vector !m o) = s(x)</td>
</tr>
<tr>
<td><code>immutable[[s, v]] = #t</code></td>
<td><code>eqall[[s, v], v]] = v</code>, where (v [[(x y) ...]]) = `eq!tab[[s, v], v, ()]]</td>
</tr>
<tr>
<td><code>eq!tab[[s, v], v, ((x y) ...)] = (#t ((x y) ...))</code></td>
<td><code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)] = (#t ((x y) ...))</code></td>
</tr>
<tr>
<td><code>where identified[[s, y, (x y) ...]]</code></td>
<td><code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)] = </code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)]`</td>
</tr>
<tr>
<td><code>where (chaperone-vector (loc x) v1, v2) = s(x)</code></td>
<td><code>where (chaperone-vector (loc x) v1, v2) = s(x)</code></td>
</tr>
<tr>
<td><code>where (impersonate-vector (loc x) v1, v2) = s(x)</code></td>
<td><code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)] = </code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)]`</td>
</tr>
<tr>
<td><code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)] = </code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)]`</td>
<td><code>where (impersonate-vector (loc x) v1, v2) = s(x)</code></td>
</tr>
<tr>
<td><code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)] = </code>eq!tab[[s, (loc x)], (loc y), ((x y) ...)]`</td>
<td><code>where (vector v ...) = s(x), (vector v ...) = s(x), \(I(v ...) = I(v ...))</code></td>
</tr>
<tr>
<td><code>where (vector-immutable v ...) = s(x), (vector-immutable v ...) = s(x), \(I(v ...) = I(v ...))</code></td>
<td><code>eq!tab[[s, v], v, ((x y) ...)] = (#t ((x y) ...))</code></td>
</tr>
<tr>
<td><code>eqs!tab[[s, v], v, ((x y) ...)]</code></td>
<td><code>eqs!tab[[s, v], v, ((x y) ...)]</code></td>
</tr>
<tr>
<td><code>where (#t ((x y) ...)) = </code>eqs!tab[[s, v], v, ((x y) ...)]`</td>
<td><code>where (#t ((x y) ...)) = </code>eqs!tab[[s, v], v, ((x y) ...)]`</td>
</tr>
<tr>
<td><code>eqs!tab[[s, (loc x)], (loc y), ((x y) ...)] = (#t ((x y) ...))</code></td>
<td><code>chaperone-of[[s, v], v]] = chaperone-of[[s, l], v2]]</code></td>
</tr>
<tr>
<td><code>where (chaperone-vector !m o) = s(x)</code></td>
<td><code>where (chaperone-vector !m o) = s(x)</code></td>
</tr>
<tr>
<td><code>chaperone-of[[s, (loc x)], (loc y)] = A[&quot;chaperone-of[[s, v], v2]]&quot;, ...]]</code></td>
<td><code>where (vector-immutable v ...) = s(x), (vector-immutable v ...) = s(x), \(I(v ...) = I(v ...))</code></td>
</tr>
<tr>
<td><code>chaperone-of[[s, v], v2]] = #t</code></td>
<td><code>chaperone-of[[s, v], v2]] = #t</code></td>
</tr>
</tbody>
</table>
5.4.4 Chaperone Erasure

To state our central theorem on chaperone proxies, we need the notion of chaperone erasure for a subset of programs. If a 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 5.4.1.** For all \( e \), if \( \text{Eval}(e) = v \) and that evaluation contains no reductions whose left hand side is \( (s_\#t\text{E}[(\text{vector-set!} (\text{loc.x}) v, n)]) \) where \( s_\#(x) = (\text{vector S} v, ...) \), then \( \text{Eval}(e_2) = v \), where \( e_2 \) is the same as \( e \) but where any uses of chaperone-vector are replaced with \( (\lambda (x y) v) \).

**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 states 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 \( (\lambda (x y) v) \), and the graph of main program allocated memory 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 of this theorem is contained in appendix B.

5.5 Performance

Although our motivation for adding chaperones and impersonators to Racket is to increase the expressiveness of the contract system, performance is major a concern. Our primary
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.

### 5.5.1 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—and 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 5.8:

- **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.
### Figure 5.8 Procedure-call microbenchmark results

<table>
<thead>
<tr>
<th>Run times in milliseconds</th>
<th>Racket</th>
<th>Chicken</th>
<th>Larceny</th>
<th>V8</th>
<th>SpiderMonkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct</td>
<td>29</td>
<td>108</td>
<td>66</td>
<td>41</td>
<td>38</td>
</tr>
<tr>
<td>indirect</td>
<td>133</td>
<td>225</td>
<td>66</td>
<td>41</td>
<td>43</td>
</tr>
<tr>
<td>wrapped</td>
<td>186</td>
<td>225</td>
<td>84</td>
<td>36</td>
<td>107</td>
</tr>
<tr>
<td>wrapped+check</td>
<td>389</td>
<td>451</td>
<td>154</td>
<td>215</td>
<td>141</td>
</tr>
<tr>
<td>wrapped+return</td>
<td>456</td>
<td>508</td>
<td>209</td>
<td>391</td>
<td>1,102</td>
</tr>
<tr>
<td>proxy</td>
<td></td>
<td></td>
<td></td>
<td>1,778</td>
<td>1,978</td>
</tr>
<tr>
<td>impersonate</td>
<td>838</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chaperone</td>
<td>846</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>impersonate+return</td>
<td>1,554</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chaperone+return</td>
<td>1,555</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>church</td>
<td>1,267</td>
<td>1,011</td>
<td>711</td>
<td>2,029</td>
<td>10,550</td>
</tr>
<tr>
<td>church-wrap</td>
<td>4,124</td>
<td>7,950</td>
<td>3,520</td>
<td>7,979</td>
<td>31,285</td>
</tr>
<tr>
<td>church-proxy</td>
<td></td>
<td></td>
<td></td>
<td>40,983</td>
<td>86,633</td>
</tr>
<tr>
<td>church-chaperone</td>
<td>46,813</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>church-chaperone/a</td>
<td>4,709</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>church-contract</td>
<td>7,537</td>
<td></td>
<td></td>
<td>297,869</td>
<td>253,380†</td>
</tr>
</tbody>
</table>

Average of three runs on MacBook Air, 1.8 GHz Intel Core i7, 4GB running OS X 10.7.3. Racket v5.3.0.1 (git 9344a7a242) 64-bit, Chicken v4.7.0 64-bit using --no-trace, Larceny v0.98b1 32-bit using -r6rs -program, V8 shell v3.10.2 (git b2b8c2c8f352) 64-bit using --harmony, SpiderMonkey v1.8.5+ (hg 638769f8ec54) 64-bit using -n -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. The source code for all benchmarks in this section is available from [http://www.cs.utah.edu/plt/chap-bm.tgz](http://www.cs.utah.edu/plt/chap-bm.tgz).

- **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.

- **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.

The results show that Racket’s performance is comparable to 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 present results for several variants in figure 5.8:

• **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 validates the practical importance of optimizations at Racket’s contract layer for the way that it generates chaperone-based contracts, in contrast to the relatively direct implementation of Contracts.coffee.

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:
5.5. PERFORMANCE

- **make guide** — Builds a representation of the Racket Guide, which involves many constructors such as `section` and `element`, as well 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.

- **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, which 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 5.9 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 for these programs.

5.5.2 Vector Performance

Our microbenchmark for vector performance is bubble sort on a vector of 10,000 integers in reverse order. Figure 5.10 shows timing results, where the “~Racket” column corresponds to
**Figure 5.9** Realistic procedure benchmark results

<table>
<thead>
<tr>
<th></th>
<th>msecs</th>
<th>makes</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>make guide</strong></td>
<td>10,750</td>
<td>9,735</td>
<td>89,461 chaperone</td>
</tr>
<tr>
<td></td>
<td>10,641</td>
<td>9,736</td>
<td>89,462 impersonate</td>
</tr>
<tr>
<td></td>
<td>10,485</td>
<td>9,283</td>
<td>88,682 no interpose</td>
</tr>
<tr>
<td></td>
<td>10,220</td>
<td>9,283</td>
<td>0 no chaperone</td>
</tr>
</tbody>
</table>

proxy overhead: 3%
additional chaperone overhead: 1%
contract checking overhead: 1%

<table>
<thead>
<tr>
<th></th>
<th>msecs</th>
<th>makes</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>render guide</strong></td>
<td>6,500</td>
<td>2,074</td>
<td>2,637,489 chaperone</td>
</tr>
<tr>
<td></td>
<td>6,453</td>
<td>2,074</td>
<td>2,637,489 impersonate</td>
</tr>
<tr>
<td></td>
<td>2,730</td>
<td>37</td>
<td>2,520,204 no interpose</td>
</tr>
<tr>
<td></td>
<td>2,552</td>
<td>37</td>
<td>0 no chaperone</td>
</tr>
</tbody>
</table>

proxy overhead: 7%
additional chaperone overhead: 1%
contract checking overhead: 136%

<table>
<thead>
<tr>
<th></th>
<th>msecs</th>
<th>makes</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>keyboard</strong></td>
<td>8,578</td>
<td>193</td>
<td>730,365 chaperone</td>
</tr>
<tr>
<td></td>
<td>8,420</td>
<td>115</td>
<td>730,285 impersonate</td>
</tr>
<tr>
<td></td>
<td>7,350</td>
<td>300</td>
<td>730,372 no interpose</td>
</tr>
<tr>
<td></td>
<td>7,208</td>
<td>76</td>
<td>0 no chaperone</td>
</tr>
</tbody>
</table>

proxy overhead: 2%
additional chaperone overhead: 2%
contract checking overhead: 15%

<table>
<thead>
<tr>
<th></th>
<th>msecs</th>
<th>makes</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>slideshow</strong></td>
<td>9,735</td>
<td>7,207</td>
<td>250,029 chaperone</td>
</tr>
<tr>
<td></td>
<td>9,609</td>
<td>7,207</td>
<td>250,032 impersonate</td>
</tr>
<tr>
<td></td>
<td>9,078</td>
<td>5,345</td>
<td>248,167 no interpose</td>
</tr>
<tr>
<td></td>
<td>9,045</td>
<td>5,345</td>
<td>0 no chaperone</td>
</tr>
</tbody>
</table>

proxy overhead: 0%
additional chaperone overhead: 1%
contract checking overhead: 6%

<table>
<thead>
<tr>
<th></th>
<th>msecs</th>
<th>makes</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>plot</strong></td>
<td>2,800</td>
<td>2,562</td>
<td>446,418 chaperone</td>
</tr>
<tr>
<td></td>
<td>2,819</td>
<td>2,562</td>
<td>446,418 impersonate</td>
</tr>
<tr>
<td></td>
<td>2,035</td>
<td>2,557</td>
<td>308,573 no interpose</td>
</tr>
<tr>
<td></td>
<td>2,017</td>
<td>2,557</td>
<td>0 no chaperone</td>
</tr>
</tbody>
</table>

proxy overhead: 1%
additional chaperone overhead: -1%
contract checking overhead: 39%

<table>
<thead>
<tr>
<th></th>
<th>msecs</th>
<th>makes</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>typecheck</strong></td>
<td>39,003</td>
<td>2,018,110</td>
<td>6,439,109 chaperone</td>
</tr>
<tr>
<td></td>
<td>38,966</td>
<td>2,018,110</td>
<td>6,439,109 impersonate</td>
</tr>
<tr>
<td></td>
<td>21,669</td>
<td>944,622</td>
<td>2,988,062 no interpose</td>
</tr>
<tr>
<td></td>
<td>21,257</td>
<td>945,064</td>
<td>0 no chaperone</td>
</tr>
</tbody>
</table>

proxy overhead: 2%
additional chaperone overhead: 0%
contract checking overhead: 80%
5.5. PERFORMANCE

Racket with specific JIT support for chaperoned vectors removed; we include this 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.

Figure 5.10 Vector microbenchmark results

<table>
<thead>
<tr>
<th></th>
<th>Racket</th>
<th>~Racket</th>
<th>Chicken</th>
<th>Larceny</th>
<th>V8</th>
<th>SpiderMonkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>1,335</td>
<td>1,335</td>
<td>3,842</td>
<td>660</td>
<td>466</td>
<td>537</td>
</tr>
<tr>
<td>proxy</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>131,574</td>
<td>135,807</td>
</tr>
<tr>
<td>chaperone</td>
<td>6,461</td>
<td>29,362</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>unsafe</td>
<td>900</td>
<td>–</td>
<td>2,628</td>
<td>825</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>unsafe*</td>
<td>697</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

~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.

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 5.10 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.

Figure 5.11 Realistic vector benchmark results

<table>
<thead>
<tr>
<th>msecs</th>
<th>makes</th>
<th>refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode-apply</td>
<td>10,747</td>
<td>1,456,221</td>
</tr>
<tr>
<td></td>
<td>10,464</td>
<td>1,456,221</td>
</tr>
<tr>
<td></td>
<td>9,751</td>
<td>1,456,231</td>
</tr>
<tr>
<td></td>
<td>7,760</td>
<td>1,456,231</td>
</tr>
<tr>
<td></td>
<td>5,974</td>
<td>219</td>
</tr>
</tbody>
</table>

vector proxy overhead: 26%
additional vector chaperone overhead: 3%
vector contract checking overhead: 7%
procedure and vector contract use overhead: 80%
To illustrate the impact on realistic programs with extensive use of vector contracts, we use the `williams/science.plt` PLaneT package. 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 5.11 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 return 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.

### 5.5.3 Structure Performance

Our microbenchmark for structure performance is to access the first field of a two-element structure 100 million times. Figure 5.12 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.

<table>
<thead>
<tr>
<th></th>
<th>Racket</th>
<th>Chicken</th>
<th>Larceny</th>
<th>V8</th>
<th>SpiderMonkey</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>direct</code></td>
<td>974</td>
<td>1,930</td>
<td>766</td>
<td>306</td>
<td>352</td>
</tr>
<tr>
<td><code>proxy</code></td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>43,253</td>
<td>23,565</td>
</tr>
<tr>
<td><code>chaperone</code></td>
<td>5,984</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><code>unsafe</code></td>
<td>293</td>
<td>116</td>
<td>743</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><code>unsafe*</code></td>
<td>291</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

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 remain unchecked. 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.

Figure 5.13 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 chaperones. All object operations required a check on the target object to determine whether it was a wrapped object, and since this test was outside the core run-time system, the JIT compiler was not able to recognize the chaperone pattern and optimize for the common case. In fact, the check interfered with optimizations that the JIT compiler could otherwise perform, and the result was a 3x slowdown on field-intensive microbenchmarks that did not use contracts [Strickland and Felleisen, 2010a]. After switching the implementation to use chaperones, this slowdown was completely eliminated.
5.5.4 Discussion

Chaperones and impersonators, as implemented in Racket, are expressive and expensive constructs—microbenchmarks indicate a factor of 5 to 10 times over normal function calls or vector or structure accesses. Despite this, real applications experience little slowdown when using them. 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. Adding interposition to Racket imposes almost no cost on the remainder of the system when interposition is not used—the only exception is a slight slowdown in unsafe operations.

- 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.

- Although Racket supports a rich hierarchy of interposition to maintain the language invariants, this additional complexity and the required dynamic checks only imposes 3% or less overhead in all of our testing. As mentioned in section 6.4, JavaScript’s proxy design limits its expressiveness by reducing the dynamic checks it performs. 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.
6.1 Contracts

Conceptually, the notion of contracts is due to Parnas [1972], who introduced it together with the notion of modules. Pragmatically, Meyer [1992a] is responsible for the terminology of “design by contracts” and the popularization of the concept in the object-oriented community. 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. Other notable examples in the Eiffel category include Euclid [Lampson et al., 1977], Ada (via Anna [Luckham and von Henke, 1985]), D [Digital Mars, 1999]; others have built Eiffel-like contract extensions for existing languages including Java, Python, Perl, and Ruby and Ciao [Mera et al., 2009]. In all of these cases, contracts are as easily implemented in the core as through pre-preprocessing because the contracts are more limited, simplistic assertions.

Technically my work follows two key pieces of research on contracts in higher-order functional programming languages. Findler and Felleisen [2002] introduced the idea of adding higher-order contracts to functional languages. Findler and Blume [2006] fleshed out the theoretical foundations of contracts by treating them as pairs of projections, which is currently the basis of implemented contract systems for higher-order languages.

6.2 Units

Flatt and Felleisen [1998] were responsible for the original unit system in Racket, where units were linked structurally. Owens and Flatt [2006] revisited the system and argued for nominal linking as the default behavior, which allows the desired link structure between units to be inferred in most cases.

All prior research on contracts assumes static boundaries between the parties of a contract and boundaries that are known at compile time. This work is the first to relax these
assumptions and to provide a framework for checking contracts in a world of higher-order
modules.

6.3 Class Contracts

In addition to the related work on contract systems described above, Findler and Felleisen
[2001] show that negative positions of contracts are contravariant and positive positions
covariant when checking contract implication between levels of a contracted class hierarchy,
which is necessary to detect violations of behavioral subtyping [Liskov and Wing, 1994].
Adding specifications for fields or methods that should not be present in a contracted class
is inspired by Bracha’s work on Jigsaw [Bracha, 1992].

6.4 Chaperones and Impersonators

The most closely related work to ours is the proxy design [Van Cutsem and Miller, 2010]
proposed for JavaScript and currently implemented in both Firefox and Chrome. These
implementations are the subject of the benchmarks in section 5. Building on the design
of mirages in AmbientTalk [Dedecker et al., 2005], 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 \( \text{===} \). 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 various
kinds of Racket values.

The JavaScript proxy API is in flux; in particular, Van Cutsem and Miller [2012] have
recently proposed a new design called direct proxies for the proxy system, which differs
significantly from the original design as implemented in current JavaScript engines.

The initial JavaScript proxy design differs most significantly from chaperones by dis-
pensing 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 points to the second major difference between JavaScript proxies and
our design: how each avoids breaking existing language invariants. JavaScript provides
few invariants that programmers may assume about the behavior of objects, due to perva-
sive 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 conditionize 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 the design of proxies considerably.

In contrast, the existing design of Racket—like that of most languages—ensures that programmers can reason using a wide variety of invariants based on information hiding, type and equality testing, plus 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. It also added implementation complexity.

These problems 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 an 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 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. This prevents JavaScript proxies from implementing higher-order contracts on immutable data, which includes any contract on methods of an immutable object. The measurements of section 5 demonstrate that these checks impose little overhead, and therefore the JavaScript design could be significantly more expressive with relatively little cost.

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. No-
table examples include the MOP [Kiczales et al., 1991], aspect-oriented programming [Kiczales et al., 1997], and java.lang.reflect.Proxy.\(^1\)

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]. Systems that enforce harmless advice by constraining side effects include Open Modules [Aldrich, 2005], EffectiveAdvice [d. S. Oliveira et al., 2010], where the latter allows constrained side effects through monads.

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 they 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 primarily ensure that invariants relating to the behavior of immutable structures are preserved.

6.4.1 Other Contract Systems and Proxies

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 contracts for opaque structures. We take this work 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 Disney’s system is not yet performant under heavy use of contracts.

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.

\(^1\)http://download.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html
Chapter 7

Conclusion

When programmers develop large systems, they use features such as first-class components to organize large code bases as the composition of small, reusable pieces. In an expressive language, these pieces can abstract over similar behavior and may be mixed and matched as desired at run-time. In the past, programmers had to choose between the use of either static components and contracts or dynamic components without protection. Similarly, contract systems either avoided mutable values or have handled them poorly.

As my dissertation finally shows, these features are compatible with contracts—provided that languages support rather basic reflection facilities. Put differently, my dissertation proves that higher-order contracts can provide effective and efficient protection of first-class components and mutable values.

7.1 Contributions

In support of my thesis, I have developed three extensions to contract systems: one for both structurally-linked and nominally-linked first-class modules; one for an object-oriented system with first-class classes, objects, and interfaces; and one for mutable values. These extensions allow programmers to create programs that use these features and to write contracts for them.

7.2 Future Work

My dissertation suggests a couple of directions for future work that would help increase the applicability and utilization of higher-order contracts.

Investigation of Other Object-Oriented Systems Some class and object systems offer even more flexibility than Racket’s. In particular, the object-oriented systems found in scripting languages such as Python and Ruby permit many more operations on classes
and objects, though this flexibility comes at the cost of making classes mutable values that can be changed arbitrarily after construction. A contract system that helps guard classes against arbitrary mutation, while allowing and appropriately checking certain changes or extensions, could help protect against misuse of such flexible object-oriented systems. I conjecture that many of my dissertation’s lessons are applicable to contract systems for such languages.

**Optimizing Contracts**  As values flow through contracted positions, they naturally accumulate contracts. In addition, components that import contracted values must be able to re-export those values with similar or stronger contracts to ensure that they are not blamed for misuse on the part of their clients. However, this contract accumulation often means rechecking the same properties of a value, over and over again. To attack this problem, we need a calculus of contracts that appropriately handles blame. This calculus would reveal possible optimizations over contract operations, which contract implementations can use to merge contracts on a given value to reduce runtime checking.
Appendix A

Proof of Blame Correctness for Section 4.3

The reduction that is responsible for deviating from well-formed terms is the evaluation of function contract monitors:

\[ E[l[mon_{jk}(\kappa_1 \ldots \kappa_n \rightarrow \kappa_r, v)]] \]
\[ \rightarrow E[l[\lambda(x_1 \ldots x_n).mon_{jk}(\kappa_r, v) \cdot mon_{jl}(\kappa_1, x_1) \ldots mon_{jl}(\kappa_n, x_n)]]] \]

This rule injects variables and applications into monitors without wrapping them with appropriate ownership annotations as the judgment for well-formed monitors requires. We can observe, though, that the variables are going to be substituted with values that have the appropriate owner because they are bound to parameters of a function whose owner is the server of the monitors around the variables. In addition, the operator of the ill-formed application has the appropriate owner if the term is well-formed before the reduction. Thus, the monitor becomes well-formed again after application.

In order to express the desired loose invariants, we equip the two judgments with an environment that keeps track of the owner of bound variables: \( l; \Gamma \vdash e \) and \( k; l; j; \Gamma \triangleright \kappa \). In most cases, the rules for terms and contracts remain the same as before except that the environment is passed around. However, the rules for terms that introduce bound variables and for monitors do change significantly. We also add new rules for intermediate expressions like guards and class values. In figure A.1, we provide the most interesting rules of the generalized judgment for well-formed terms.

According to figure A.2, the rule for \( mon_{jk}(\kappa, e) \) delegates checking of the monitor expression to the auxiliary judgment \( l; \Gamma \vdash e \). These additional rules allow for expressions in monitors to be either a term or a variable with the appropriate owner or an application where the operator term has the appropriate owner.

The new judgment generalizes the initial one.

**Proposition A.0.1.** If \( l \vdash e \) then \( l; \emptyset \vdash e \).

**Proof Idea.** We proceed by induction on the height of the derivation of \( l \vdash e \).
Figure A.1 Ownership coincides with contract monitors (2)

<table>
<thead>
<tr>
<th>$l; \Gamma \vdash e$</th>
<th>$l; \Gamma \vdash \text{error}^k$</th>
<th>$l; \Gamma \vdash \text{object}(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l; \Gamma \vdash l_1 ; \ldots ; \ldots ; e$</td>
<td>$l; \Gamma \vdash l_1 \text{isend}^d(e_o, m, e_1 \ldots)$</td>
<td>$l; \Gamma \vdash \text{super}^t(e_o, m, e_1 \ldots)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$l; \Gamma \vdash e$</th>
<th>$k; \Gamma \vdash e$</th>
<th>$k; l; j; \Gamma \vdash \kappa$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l; \Gamma \vdash \lambda(x_1 \ldots).e$</td>
<td>$l; \Gamma \vdash e_o \quad l; \Gamma \vdash e_1 \ldots$</td>
<td>$l; \Gamma \vdash \text{mon}^{k,l}(\kappa, e)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$l; \Gamma \vdash e_x$</th>
<th>$l; \Gamma \vdash \text{class} { e_x }$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l; \Gamma \vdash \text{inherit} m_{i_1} \ldots$</td>
<td>$l; \Gamma \vdash \text{public} { m_{p_1}(\text{this}<em>{p_1}, x</em>{p_1}^1 \ldots) e_{p_1} \ldots }$</td>
</tr>
<tr>
<td>$l; \Gamma \vdash \text{override} { m_{o_1}(\text{this}<em>{o_1}, x</em>{o_1}^1 \ldots) e_{o_1} \ldots }$</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.2 Ownership is preserved for monitored expressions

<table>
<thead>
<tr>
<th>$l; \Gamma \vdash e$</th>
<th>$k; l; j; \Gamma \vdash \kappa_{p_1} \rightarrow \kappa_{p_1} \rightarrow \kappa_{r_1} \rightarrow \kappa_{r_1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l; \Gamma \vdash v$</td>
<td>$l; \Gamma \vdash \text{public} { m_{p_1}(\kappa_{p_1} \rightarrow \kappa_{r_1} \rightarrow \kappa_{r_1} \rightarrow \kappa_{r_1})^k,l \ldots }$</td>
</tr>
<tr>
<td>$l; \Gamma \vdash \text{override} { m_{o_1}(\kappa_{o_1} \rightarrow \kappa_{r_1} \rightarrow \kappa_{r_1} \rightarrow \kappa_{r_1})^k,l \ldots }$</td>
<td></td>
</tr>
</tbody>
</table>

With $l; \Gamma \vdash e$, we can prove the central subject reduction lemma.

Main Lemma A.0.2. If $l_o; \emptyset \vdash e$ and $e \rightleftharpoons e_0$, then $l_o; \emptyset \vdash e_0$.

Proof. We proceed by case analysis on the reduction of $e$. The cases for functional CFCC are identical to the proof of theorem 5 of Dimoulas et al. [2011]. Here we present only the cases that correspond to the class features of CFCC.

- $E'[\text{new}(v)] \rightarrow E'[\text{object}(v)]$

By assumption $l_o; \emptyset \vdash e$, for which lemma A.0.3 implies that $l; \emptyset \vdash \text{new}(v)$. The inference rules imply $l; \emptyset \vdash v$. We can use the same label to check via the inference rules that $l; \emptyset \vdash \text{object}(v)$. Hence, $l_o; \emptyset \vdash E'[\text{object}(v)]$. 
where $c = \text{class}\{v_s$
\begin{itemize}
  \item inherit $m_{i_2}$
  \item public $[m_{p_1}(\text{this}_{p_1}, x_1^{p_1}, \ldots) e_{p_1}]$
  \item override $[m_{o_1}(\text{this}_{o_1}, x_1^{o_1}, \ldots) e_{o_1}]$
\end{itemize}

and $cv = \text{class}/v^i\{v_s$
\begin{itemize}
  \item methods $[m_{p_1}(\text{this}_{p_1}, x_1^{p_1}, \ldots) \{l/\text{id}\} e_{p_1}]$
  \item $[m_{o_1}(\text{this}_{o_1}, x_1^{o_1}, \ldots) \{l/\text{id}\} e_{o_1}]$
\end{itemize}

Using the assumptions and lemma A.0.3, we get $l; \emptyset \vdash c$. Thus, by the inference rules, $l; \emptyset \vdash v_s$, l; $\{\text{this}_{p_1} : l, x_1^{p_1} : l, \ldots \} \vdash e_{p_1}, \ldots$ and $l; \{\text{this}_{o_1} : l, x_1^{o_1} : l, \ldots \} \vdash e_{o_1}, \ldots$. By lemma A.0.4, we get $l; \{\text{this}_{p_1} : l, x_1^{p_1} : l, \ldots \} \vdash \{l/\text{id}\} e_{p_1}, \ldots$ and $l; \{\text{this}_{o_1} : l, x_1^{o_1} : l, \ldots \} \vdash \{l/\text{id}\} e_{o_1}, \ldots$. With an application of the inference rules for well-formed expressions, we obtain $l; \emptyset \vdash cv$ and $l_o; \emptyset \vdash E^i[cv]$.

- $E^i[\text{send}(v_o, m, v \ldots)] \rightarrow E^i[\text{Pull}[v_o, m] v_o v_1 \ldots v_n]$

By the assumptions and lemma A.0.3, we conclude $l; \emptyset \vdash \text{send}(v_o, m, v \ldots)$. Thus, by the inference rules, $l; \emptyset \vdash v_1$, l; $\emptyset \vdash v_n$, and $l; \emptyset \vdash v_o$ and by lemma A.0.5, $l; \emptyset \vdash \text{Pull}[v_o, m]$. Finally by the inference rules, we get $l; \emptyset \vdash \text{Pull}[v_o, m] v_o v_1 \ldots v_n$ and we obtain $l_o; \emptyset \vdash E^i[\text{Pull}[v_o, m] v_o v_1 \ldots v_n]$.

- $E^i[\text{super}^i(v_o, m, v_1 \ldots v_n)] \rightarrow E^i[\text{Pull}[\text{GetS}[v_o, t, l], m] v_o v_1 \ldots v_n]$

By the usual chain of thoughts, we obtain $l; \emptyset \vdash v_1$, l; $\emptyset \vdash v_n$, and $l; \emptyset \vdash v_o$. Lemmas A.0.6 and A.0.5 give us $l; \emptyset \vdash \text{Pull}[\text{GetS}[v_o, t, l], m]$. Applying the inference rules for well-formed expressions, we conclude $l; \emptyset \vdash \text{Pull}[\text{GetS}[v_o, t, l], m] v_o v_1 \ldots v_n$ and $l_o; \emptyset \vdash E^i[\text{Pull}[\text{GetS}[v_o, t, l], m] v_o v_1 \ldots v_n]$.

- $E^i[\text{isend}(v_o, m, v_1 \ldots v_n)] \rightarrow E^i[\text{Find}[v_o, t, m, l] v_o v_1 \ldots v_n]$

As in the previous cases, we get the following: $l; \emptyset \vdash v_1$, l; $\emptyset \vdash v_n$, and $l; \emptyset \vdash v_o$. By lemma A.0.7, we obtain $l; \emptyset \vdash \text{Find}[v_o, t, m, l]$. We put all the pieces together with the inference rules for well-formed expressions to get $l; \emptyset \vdash \text{Find}[v_o, t, m, l] v_o v_1 \ldots v_n$ and finally, $l_o; \emptyset \vdash E^i[\text{Find}[v_o, t, m, l] v_o v_1 \ldots v_n]$.

- $E^i[\text{mon}^{k,l}(k, v)] \rightarrow E^i[v']$

where $\kappa = \text{class}/c\{\text{public} [m_{p_1}(\kappa_{p_1}^{p_1}, \kappa_{1}^{p_1}, \ldots) \Rightarrow \kappa_{p_1}^{p_1}])$
\begin{itemize}
  \item override $[m_{o_1}(\kappa_{o_1}^{o_1}, \kappa_{1}^{o_1}, \ldots) \Rightarrow \kappa_{o_1}^{o_1}])$
\end{itemize}

and
\[ v' = G\{ v \]

\[
\text{public } [m_{p_1} (\kappa_{i_1}^{p_1} \kappa_{j_1}^{p_1} \cdots \kappa_{i_j}^{p_j} \cdots)] \]

\[
\text{override } [m_{o_1} (\kappa_{i_1}^{o_1} \kappa_{j_1}^{o_1} \cdots \kappa_{i_j}^{o_j} \cdots)] \]

Similar to the previous case, we deduce that \( v = \|v_0\|^k \), \( k; \emptyset \vdash v_0 \), and \( l; \emptyset \vdash \text{mon}_{j}^{k,l}(\kappa, v) \). From the contract well-formedness rules we acquire the following:

\[
l; k; j; \emptyset \triangleright \kappa_{i_1}^{p_1} \kappa_{j_1}^{p_1} \cdots \kappa_{i_j}^{p_j} \cdots \]

and

\[
k; l; j; \emptyset \triangleright \kappa_{i_1}^{o_1} \kappa_{j_1}^{o_1} \cdots \kappa_{i_j}^{o_j} \cdots \]

We combine the above pieces of information with the well-formedness rules and get the desired conclusion, \( \Gamma; l \vdash v' \).

- \( E'[\text{mon}_{j}^{k,l}(\text{any}, v)] \rightarrow E'[v] \)

As usual, we deduce that \( v = \|v_0\|^k \), \( k; \emptyset \vdash v_0 \), and \( l; \emptyset \vdash \text{mon}_{j}^{k,l}(\text{any}, v) \). We combine the above pieces of information with the well-formedness rules and get the desired conclusion, \( \Gamma; l \vdash v \).

The proof of the main lemma A.0.2 requires several lemmas regarding properties of the evaluation contexts and the meta-functions with respect to the judgment for well-formed terms. We list them and their proof ideas below.

**Lemma A.0.3.** If \( l; \emptyset \vdash E^k[e] \) then \( k; \emptyset \vdash e \).

**Proof Idea.** By induction on the size of \( E^k \).

**Lemma A.0.4.** If \( l; \Gamma \vdash e \), \( l; \Gamma \vdash \{i/\text{id}\} \).

**Proof Idea.** By induction on the height of \( k; \Gamma \vdash e \).

**Lemma A.0.5.** If \( k; \emptyset \vdash \|\gamma\| \cdot \), \( m \in \text{Methods}[\|\gamma\| \cdot ] \), and \( CV[\|\gamma\| \cdot ] \) or \( \text{Object}[\|\gamma\| \cdot ] \), \( k; \emptyset \vdash \text{Pull}[\|\gamma\| \cdot , m] \).

**Proof Idea.** By induction on the height of \( k; \emptyset \vdash \|\gamma\| \cdot \).

**Lemma A.0.6.** If \( l; \emptyset \vdash \|\gamma\| \cdot , \text{Is}[\|\gamma\| \cdot , m] \), and \( CV[\|\gamma\| \cdot ] \) or \( \text{Object}[\|\gamma\| \cdot ] \), for all \( k \), \( k; \emptyset \vdash \text{GetS}[\|\gamma\| \cdot , i, l] \) and \( CV[\text{GetS}[\|\gamma\| \cdot , i, l]] \).

**Proof Idea.** By induction on the height of \( l; \emptyset \vdash \|\gamma\| \cdot \).

**Lemma A.0.7.** If \( l; \emptyset \vdash \|\gamma\| \cdot , m \in \text{Methods}[\text{Get}[\|\gamma\| \cdot , i, l]], \text{Is}[\|\gamma\| \cdot , m] \), and \( CV[\|\gamma\| \cdot ] \) or \( \text{Object}[\|\gamma\| \cdot ] \), for all \( k \), \( k; \emptyset \vdash \text{Find}[\|\gamma\| \cdot , i, m, l] \).
Proof Idea. By induction on the height of \( l; \emptyset \vdash \|\gamma\|^{\bar{I}} \) using lemmas A.0.5 and A.0.8 for the two base cases.

Lemma A.0.8. If \( l; \emptyset \vdash \|\gamma\|^{\bar{I}}, \mathcal{I}s[\|\gamma\|^{\bar{I}}, l], \mathcal{C}V[\|\gamma\|^{\bar{I}}], m \in \mathcal{M}e\text{ethods}[\|\gamma\|^{\bar{I}}, l] \), \( v = \|v_o\|^{\bar{t}} \otimes \bar{I} \), and \( l \otimes \bar{I}; \emptyset \vdash \|v_o\| \), for all \( k \), \( k; \emptyset \vdash \mathcal{P}ush[\|\gamma\|^{\bar{I}}, \bar{v}, \bar{t}, m, l] \).

Proof Idea. By induction on the height of \( l; \emptyset \vdash \|\gamma\|^{\bar{I}} \).

The main lemma A.0.2 is all we need to show that CFCC is blame correct.

Theorem A.0.9. \( \longrightarrow \) is blame correct.

Proof. From the proof of the subject reduction theorem, we know that the subject implies

\[ l_o; \emptyset \vdash \mathcal{E}[\|\gamma\|^{\bar{I}}, \text{mon}^k, \text{flat}(e_c)^k, \|v\|^k] \]

The owner of the hole of the context is the same as the client label, and the owner of the monitored value \( v \) is the same as the server label. In addition, the term in the hole is well-formed. Hence, by lemma A.0.3 the contract is part of the obligations of the server and carries the server’s label.
APPENDIX B

Proof of Chaperone Erasure for Section 5.4.4

The definition of the approximates relation $<e_1, s_1> \sim <e_2, s_2>$ is given at the end of this appendix.

Lemma B.1 (Substitution lemma):
For all $e_1, s_1, e_2, s_2, x \ldots, e_3 \ldots, \text{and } e_4 \ldots$
if $<e_1, s_1> \sim <e_2, s_2>$ and $<e_3, s_1> \sim <e_4, s_4>$ ...
then $<e_1[x \mapsto e_3, \ldots], s_1> \sim <e_2[x \mapsto e_4, \ldots], s_2>$.

Lemma B.2 (approximations of unique decomposition):
For all $e_1, s_1, e_2, s_2$.
if $<e_1, s_1> \sim <e_2, s_2>$ and $e_2 = E_2[e_4]$,
then $e_1 = E_1[e_3], <E_1, s_1> \sim <E_2, s_2>$, and $<e_3, s_1> \sim <e_4, s_2>$.

Lemma B.3 (context filling honors approximation):
For all $E_1, e_1, s_1, E_2, e_2, s_2$.
if $<e_1, s_1> \sim <e_2, s_2>$ and $<E_1, s_1> \sim <E_2, s_2>$,
then $<E_1[e_1], s_1> \sim <E_2[e_2], s_2>$.

General argument for the next four lemmas: approximation ensures that the combination of a value and a store has the same graph structure (ignoring chaperones) as its approximate value/store. Thus, the traversal of that graph structure done by immutable, equal, and chaperone-of will reveal the same result on the approximate value/store as the original value/store, since addition or removal of chaperones does not affect the result of these operations.
Lemma B.4 (approximations are likewise equal):
For all \(v_1, v_3, s_1, e_2, v_4, s_2\).
if \(<v_1, s_1> \sim <v_2, s_2>\) and \(<v_3, s_1> \sim <v_4, s_4>\),
then equal[[s_1, v_1, v_3]] = equal[[s_2, v_2, v_4]].

Lemma B.5 (approximations are likewise immutable):
For all \(v_1, s_1, v_2, s_2\).
if \(<v_1, s_1> \sim <v_2, s_2>\),
then immutable[[s_1, v_1]] = immutable[[s_2, v_2]].

Lemma B.6 (approximations are likewise chaperone-of):
For all \(v_1, v_3, s_1, v_2, v_4, s_2\).
if \(<v_1, s_1> \sim <v_2, s_2>\) and \(<v_3, s_1> \sim <v_4, s_4>\),
then chaperone-of[[s_1, v_1, v_3]] = chaperone-of[[s_2, v_2, v_4]].

Lemma B.7 (chaperones of approximates are approximates):
For all \(v_1, v_3, s_1, v_2, s_2\):
If chaperone-of[[s_1, v_3, v_1]] and \(<v_1, s_1> \sim <v_2, s_2>\),
then \(<v_3, s_1> \sim <v_2, s_2>\).

(The \(\sim\) relation strips off chaperones when it finds them when checking approximation, so adding one doesn’t change the result.)

Lemma B.8 (approximates are still approximates in pure store extensions):
For all \(v_1, s_1, s_1', v_2, s_2, s_2'\):
If \(<v_1, s_1> \sim <v_2, s_2>\),
and \(s_1' \sim s_1\),
and \(s_2' \sim s_2\),
then \(<v_1, s_1'> \sim <v_2, s_2'>\).

(Define \(\sim\) to be the same as \(\leq\) except with the additional caveat that if \(s_1(x) = (\text{vector} \ #f \ v \ ...),\) then \(s_1'(x) = (\text{vector} \ #f \ v \ ...).\)
That is, there are no changes in vectors allocated by the main program.
Since no vectors of the form \((\text{vector} \ #t \ v \ ...)\) are traversed by a successful approximation, the approximation algorithm will follow exactly the same path with the same results in the extensions.)
Lemma B.9:

For all $e_2$ that do not contain set-marker, get-marker, or chaperone-vector,
let $E_2[e_6] = e_2$.

If there exists an $s_2$, $v_2$, $s_4$.
$\langle E_2[e_6], #f, s_2 \rangle$ reduces to $\langle E_2[v_2], #f, s_4 \rangle$.

For all $e_1$ and $s_1$ such that $\langle e_1, s_1 \rangle \sim \langle e_2, s_2 \rangle$, let $E_1[e_5] = e_1$.
Also, require that the reduction of $\langle e_1, #f, s_1 \rangle$ contains no program states of the form $\langle E[(vector-set! (loc x) n v), #t, s \rangle$ where $s(x) = (vector #f v_e ...)$.

Either:

1) $\langle e_1, #f, s_1 \rangle$ diverges
2) there exists a $b$, $s_3$.
   $\langle e_1, #f, s_1 \rangle$ reduces to $\langle(error 'variable), b, s_3 \rangle$
3) there exists an $e_3$, $b$, $s_3$.
   $\langle e_1, #f, s_1 \rangle$ reduces to $\langle e_3, b, s_3 \rangle$ and $e_3$ is a stuck state.
4) there exists a $v_1$, $s_3$.
   $\langle e_1, #f, s_1 \rangle$ reduces to $\langle E_1[v_1], #f, s_3 \rangle$ and $\langle E_1[v_1], s_3 \rangle \sim \langle E_2[v_2], s_4 \rangle$.

Proof:

Fix $e_2 = E_2[e_6]$. Retrieve $s_2$, $v_2$, $s_4$ from our hypothesis about reduction (I), and fix $e_1$ and $s_1$. Since we have that $e_1$ and $e_2$ are approximates in their respective stores (hypothesis II), we know from Lemma B.2 that $\langle E_1, s_1 \rangle \sim \langle E_2, s_2 \rangle$ and $\langle E_5, s_1 \rangle \sim \langle E_6, s_2 \rangle$.

Now we'll induct on the length of the reduction sequence from $\langle E_2[e_6], #f, s_2 \rangle$ to $\langle E_2[v_2], #f, s_4 \rangle$ and the size of the chaperone chain (if any) for values in $e_5$. That is, either we make progress by taking a step in $e_2$, or we make progress by removing a chaperone from some part of the values present in $e_5$.

$\langle E_2[(lambda (y \ldots) e_b)], #f, s_2 \rangle \rightarrow \langle E_2[(loc z)], #f, s_2[z \rightarrow e_6] \rangle$
Since $\langle e_5, s_1 \rangle \sim \langle e_6, s_2 \rangle$, then $e_5$ is either a lambda term or is the operator ‘chaperone-vector’ (if $e_6$ is $(\lambda (v x y) v)$).

If $e_5 = \text{chaperone-vector}$,

$\langle e_1, #f, s_1 \rangle \sim \langle E_2[(\text{loc } z)], #f, s_2[z \rightarrow e_6] \rangle$. Applying the IH on the rest of the reduction sequence, using $e_1$ and $s_1$ and the approximation above to discharge the hypothesis, we get our desired result immediately.

If $e_5$ is a lambda, then we choose a fresh location $w$.

$\langle E_1[e_5], #f, s_1 \rangle \rightarrow \langle E_1[(\text{loc } w)], #f, s_1[w \rightarrow e_5] \rangle$

By the definition of the approximation location,

$\langle E_1[(\text{loc } w)], #f, s_1[w \rightarrow e_5] \rangle \sim \langle E_2[(\text{loc } z)], #f, s_2[z \rightarrow e_6] \rangle$,

since in each case we’re just introducing an indirection into the store.

Since the two locations are fresh, we won’t run into the case where one appears in the mapping but the other doesn’t, and the locations point to approximately equal values (the store addition doesn’t change their approximateness). Applying the IH to the rest of the reduction sequence, using $E_1[(\text{loc } w)]$ and $s_1[w \rightarrow e_5]$ as our new $e_1$ and $s_1$ and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old $e_1$ if we don’t get divergence, a stuck state, or an error. If we do, then $e_1$ diverges, gets stuck, or errors, respectively.

$\langle E_2[((\text{loc } z) v_4 \ldots)], #f, s_2 \rangle \rightarrow \langle E_2[e_b2[y_2 \rightarrow v_4, \ldots]], #f, s_2 \rangle$

where $s_2(z) = (\lambda (y_2 \ldots) e_b2)$

There are two cases for $e_5$ from the definition of approximation:

e_5 = ((\text{loc } w) v_3 \ldots)

Since $\langle E_1[((\text{loc } w) v_3 \ldots)], s_1 \rangle \sim \langle E_2[((\text{loc } z) v_3 \ldots)], #f, s_2 \rangle$ and $s_2(z) = (\lambda (y_2 \ldots) e_b2)$,

then from approximation we get $s_1(w) = (\lambda (y_1 \ldots) e_b1)$, where $\langle (\lambda (y_1 \ldots) e_b1), s_1 \rangle \sim \langle (\lambda (y_2 \ldots) e_b2), s_2 \rangle$.

Since $\langle v_3, s_1 \rangle \sim \langle v_4, s_2 \rangle$ for each $v_3$ and $v_4$,

$\langle E_1[e_b1[y_1 \rightarrow v_3, \ldots]], s_1 \rangle \sim \langle E_2[e_b2[y_2 \rightarrow v_4, \ldots]], s_2 \rangle$.

Apply our IH to the rest of the reduction sequence for $e_2$, using $E_1[e_b1[y_1 \rightarrow v_3, \ldots]]$ and $s_1$ as $e_1$ and $s_1$ and the approximation above to discharge the hypothesis, and stitch together the reduction
sequence we get back with the step we took above in e_1.

\[ e_5 = (\text{chaperone-vector } v_3 \ v_5 \ v_7): \]

Then \[ e_6 = ((\text{loc } z) \ v_4 \ v_6 \ v_8) \] and \[ s_2(z) = (\lambda v (v \ x \ y) \ v), \]

thus the RHS of the reduction step for e_2 simplifies to

\[ < E_2[v_4], \ #f, s_2 >. \]

We know that \( < v_3, s_1 > \sim < v_4, s_2 > \) from hypothesis II.

Our first step in the reduction of e_1 is:

\[ < E_1[(\text{chaperone-vector } v_3 \ v_5 \ v_7)], \ #f, s_1 > \rightarrow \]

\[ < E_1[(\text{loc } w)], \ #f, s_1[w \rightarrow] (\text{chaperone-vector } v_3 \ v_5 \ v_7)] > \]

Since \( \sim \) ignores chaperones, (\text{loc } w) points to a chaperone of \( v_3 \), and

the new store just adds a new mapping and doesn’t change old ones,

we have that

\[ < E_1[(\text{loc } w)], \ s_1[w \rightarrow] (\text{chaperone-vector } v_3 \ v_5 \ v_7)] > \sim \]

\[ < E_2[v_4], \ s_2 >. \]

Apply the IH to the rest of the reduction sequence for e_2, using

the LHS of the approximation above as our new e_1 and s_1, and then

stitch together the results with the single step taken above.

\[ < E_2[(\text{error 'variable})], \ #f, s_2 > \rightarrow <(\text{error 'variable}), \ #f, s_2 > \]

Breaks the hypothesis that \( e_2, \ #f, s_2 \) reduces to \( < E_2[v_2], \ #f, s_4 >. \)

\[ < E_2[(\text{vector } v_4 ...)], \ #f, s_2 > \rightarrow \]

\[ < E_2[(\text{loc } z)], \ #f, s_2[z \rightarrow] (\text{vector } #f \ v_4 ...)] > \]

\[ e_1 = E_1[(\text{vector } v_3 ...)], \text{ so we can take a step} \]

\[ < E_1[(\text{vector } v_3 ...)], \ #f, s_1 > \rightarrow \]

\[ < E_1[(\text{loc } w), \ #f, s_1[w \rightarrow] (\text{vector } #f \ v_3 ...)] > \]

From the approximation hypothesis, we have that \( < v_3, s_1 > \sim < v_4, s_2 > \)

for all \( v_3 \) and \( v_4 \). By the definition of the approximation location,

\[ < E_1[(\text{loc } w)], \ #f, s_1[w \rightarrow] (\text{vector } #f \ v_3 ...)] > \sim \]

\[ < E_2[(\text{loc } z)], \ #f, s_2[z \rightarrow] (\text{vector } #f \ v_4 ...)] >, \]

since in each case we’re just introducing an indirection into the store

(plus adding the boolean that marks when this vector was allocated).

Since the two locations are fresh, we won’t run into the case where one
appears in the mapping but the other doesn’t, and the locations point to approximately equal values (the store addition doesn't change their approximateness). Applying the IH to the rest of the reduction sequence, using E_1[(loc w)] and s_1[w |-> (vector #f v_3 ...)] as our new e_1 and s_1 and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old e_1. As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

\[\langle E_2[(\text{vector-immutable } v_4 \ldots)], \#f, s_2 \rangle \rightarrow\]
\[\langle E_2[(\text{loc } z)], \#f, s_2[z |-> (\text{vector-immutable } v_4 \ldots)]\rangle\]

\[e_1 = E_1[(\text{vector-immutable } v_3 \ldots)], \text{ so we can take a step}\]
\[\langle E_1[(\text{vector-immutable } v_3 \ldots)], \#f, s_1 \rangle \rightarrow\]
\[\langle E_1[(\text{loc } w), \#f, s_1[w |-> (\text{vector-immutable } v_3 \ldots)]\rangle\]

From the approximation hypothesis, we have that \(v_3, s_1 \sim v_4, s_2\) for all \(v_3\) and \(v_4\). By the definition of the approximation location,
\[\langle E_1[(\text{loc } w)], \#f, s_1[w |-> (\text{vector-immutable } v_3 \ldots)]\rangle \sim\]
\[\langle E_2[(\text{loc } z)], \#f, s_2[z |-> (\text{vector-immutable } v_4 \ldots)]\rangle,\]
since in each case we’re just introducing an indirection into the store.

Since the two locations are fresh, we won’t run into the case where one appears in the mapping but the other doesn’t, and the locations point to approximately equal values (the store addition doesn't change their approximateness). Applying the IH to the rest of the reduction sequence, using E_1[(loc w)] and s_1[w |-> (vector-immutable v_3 ...)] as our new e_1 and s_1 and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old e_1. As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

\[\langle E_2[(\text{impersonate-vector } l_2 m_2 o_2)], \#f, s_2 \rangle \rightarrow\]
\[\langle E_2[(\text{loc } z)], \#f, s_2[z |-> (\text{impersonate-vector } l_2 m_2 o_2)]\rangle\]

\[e_1 = E_1[(\text{impersonate-vector } l_1 m_1 o_1)], \text{ so we can take a step}\]
\[\langle E_1[(\text{impersonate-vector } l_1 m_1 o_1)], \#f, s_1 \rangle \rightarrow\]
\[\langle E_1[(\text{loc } w), \#f, s_1[w |-> (\text{impersonate-vector } l_1 m_1 o_1)]\rangle\]
From the approximation hypothesis, we have that \( <l_1, s_1> \sim <l_2, s_2> \), \( <m_1, s_1> \sim <m_2, s_2> \), and \( <o_1, s_1> \sim <o_2, s_2> \).

By the definition of the approximation location,

\[
\begin{align*}
&E_1[(\text{loc } w)], \#f, s_1[w \mapsto (\text{impersonate-vector } l_1 m_1 o_1)] > \\
&\sim \\
&E_2[(\text{loc } z)], \#f, s_2[z \mapsto (\text{impersonate-vector } l_2 m_2 o_2)] >
\end{align*}
\]

since in each case we’re just introducing an indirection into the store. Since the two locations are fresh, we won’t run into the case where one appears in the mapping but the other doesn’t, and the locations point to approximately equal values (the store addition doesn’t change their approximateness). Applying the IH to the rest of the reduction sequence, using \( E_1[(\text{loc } w)] \) and \( s_1[w \mapsto (\text{impersonate-vector } l_1 m_1 o_1)] \) as our new \( e_1 \) and \( s_1 \) and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old \( e_1 \). As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

immutable?, equal?, and chaperone-of? cases

Follows from the lemmas about immutable, equal, and chaperone-of on approximations above.

\[
\begin{align*}
&E_2[(\text{immutable? } v_4)], \#f, s_2 > \sim \langle E_2[\text{immutable}[[s_2, v_4]]], \#f, s_2 > \\
\end{align*}
\]

\( e_1 = E_1[(\text{immutable? } v_3)] \), so we can take a step

\[
\begin{align*}
&E_1[(\text{immutable? } v_3)], \#f, s_1 > \sim \langle E_1[\text{immutable}[[s_1, v_3]]], \#f, s_1 > \\
\end{align*}
\]

From approximation, we get \( <v_3, s_1> \sim <v_4, s_2> \), and from lemma B.5, we get that \( \text{immutable}[[s_1, v_3]] = \text{immutable}[[s_2, v_4]] \), so

\[
\begin{align*}
&E_1[\text{immutable}[[s_1, v_3]]], s_1 > \sim \langle E_2[\text{immutable}[[s_2, v_4]]], s_2 >
\end{align*}
\]

Applying the IH to the rest of the reduction sequence, using \( E_1[(\text{immutable? } v_3)] \) and \( s_1 \) as our new \( e_1 \) and \( s_1 \) and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old \( e_1 \). As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

Applications of chaperone-of? and equal?:

Follows similarly using the appropriate lemma.
<E_2[(vector-ref (loc z) n), #f, s_2] ->
  <E_2[(m_2 l_2 n (vector-ref 1_2 n))], #f, s_2>
where s_2(z) = (impersonate-vector 1_2 m_2 o_2)

\[ e_1 = E_1[(vector-ref (loc w) n)], \]
but based on the approximation from hypothesis II, there are two possibilities for s_1(w):

\[ s_1(w) = (impersonate-vector l_1 m_1 o_1) \]

Then we get the following reduction step:
\[ <E_1[(vector-ref (loc w) n)], #f, s_1] -> \]
\[ <E_1[(m_1 l_1 n (vector-ref l_1 n))], #f, s_1> \]
and
\[ <E_1[(m_1 l_1 n (vector-ref l_1 n))], s_1> \]
\[ <E_2[(m_2 l_2 n (vector-ref 1_2 n))], #f, s_2> \].

Applying the IH to the rest of the reduction sequence, using \( E_1[(m_1 l_1 n (vector-ref l_1 n))] \) and \( s_1 \) as our new \( e_1 \) and \( s_1 \) and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old \( e_1 \). As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

\[ s_1(w) = (chaperone-vector l_1 m_1 o_1) \]

Then we get the following reduction step:
\[ <E_1[(vector-ref (loc w) n)], #f, s_1] -> \]
\[ <E_1[(let ([old (vector-ref l_1 n)])
  (let ([new (set-marker (m_1 l_1 n old))]
    (clear-marker (if (chaperone-of? new old)
      new
      (error 'bad-cvref))))))],
  #f, s_1> \]

Due to the approximation relation, we know that
\[ <l_1, s_1> \sim <(loc z), s_1> \] (since we skip through chaperones).
So what we will do is use the entire reduction for \( e_6 \), but use \( E_1[(vector-ref l_1 n)] \) as \( e_1 \) (and keep \( s_1 \) the same), which removes the calculation of a chaperone. From our IH, we get that
<E_1[(vector-ref l_1 n)], #f, s_1> either:

* Diverges: then the reduction of e_1 diverges
* Errors: then the reduction of e_1 errors
* Reaches a stuck state: then the reduction of e_1 reaches a stuck state.
* Reduces to <E_1[v_1], #f, s_3'> for some v_1' and s_3'
  where <v_1', s_3'> \sim <v_2, s_4>.

Then in reducing e_1, we get the same steps in the RHS of the first let:

<E_1[(vector-ref (loc w) n)], #f, s_1> \rightarrow
  \rightarrow
  \rightarrow

For reducing (m_1 l_1 n v_1') in this context, there are several cases:
* <E_1[(let ([old v_1'])

  (let ([new (set-marker (m_1 l_1 n old))])
    (clear-marker (if (chaperone-of? new old)
      new
      (error 'bad-cvref)))))],

  #f, s_3'> \rightarrow

* <E_1[(let ([new (set-marker (m_1 l_1 n v_1'))])

  (clear-marker (if (chaperone-of? new v_1')
    new
    (error 'bad-cvref)))))],

  #f, s_3'> \rightarrow

* <E_1[(let ([new (m_1 l_1 n v_1')])

  (clear-marker (if (chaperone-of? new v_1')
    new
    (error 'bad-cvref)))))],

  #t, s_3'>

* <E_1[(let ([new (m_1 l_1 n v_1')])

  (clear-marker (if (chaperone-of? new v_1')
    new
    (error 'bad-cvref)))))],

  #t, s_3'> diverges: then reducing e_1 diverges

* <E_1[(let ([new (m_1 l_1 n v_1')])

  (clear-marker (if (chaperone-of? new v_1')
    new
    (error 'bad-cvref)))))],

  #t, s_3'> diverges: then reducing e_1 diverges
APPENDIX B. PROOF: CHAPERONE ERASURE

* Not a chaperone: then the reduction of e_1 errors.

* Is a chaperone. Then we have chaperone-of[[s_3'', v_1'', v_1']], and
  \[<E_1[(if (chaperone-of? v_1'' v_1') v_1'' (error 'bad-cvref))], #f, s_3'']\]

Now there are two cases: v_1'' is not a chaperone of v_1' or it is.

* Not a chaperone: then the reduction of e_1 errors.

* Is a chaperone. Then we have chaperone-of[[s_3'', v_1'', v_1']], and
  \[<E_1[(if (chaperone-of? v_1'' v_1') v_1'' (error 'bad-cvref))], #f, s_3'']\]
s_3' <= s_3'', and because of the restrictions on the reduction of e_1, s_3' \sim s_3''. Therefore, \langle v_1', s_3'' \rangle \sim \langle v_2, s_2 \rangle by lemma B.8 and by lemma B.7, \langle v_1'', s_3'' \rangle \sim \langle v_2, s_2 \rangle.

Therefore v_1'' is the v_1 we need, and s_3'' is the s_3 we need to finish this case.

\langle E_2[(vector-set! (loc z) n v_4), #f, s_2] \rangle
\langle E_2[(vector-set! l_2 n (o_2 l_2 n v_4))], #f, s_2 \rangle
where s_2(z) = (impersonate-vector l_2 m_2 o_2)

e_1 = E_1[(vector-set! (loc w) n v_3)], but based on the approximation from hypothesis II, there are two possibilities for s_1(w):

s_1(w) = (impersonate-vector l_1 m_1 o_1)
Then we get the following reduction step:
\langle E_1[(vector-set! (loc w) n)], #f, s_1 \rangle \rightarrow
\langle E_1[(vector-set! l_1 n (o_1 l_1 n v_3))], #f, s_1 \rangle
and \langle E_1[(vector-set! l_1 n (o_1 l_1 n v_3))], #f, s_1 \rangle \sim
\langle E_2[(vector-set! l_2 n (o_2 l_2 n v_4))], #f, s_2 \rangle

Applying the IH to the rest of the reduction sequence, using E_1[(vector-set! l_1 n (o_1 l_1 n v_3))] and s_1 as our new e_1 and s_1 and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old e_1. As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

s_1(w) = (chaperone-vector l_1 m_1 o_1)
Then we get the following reduction step:
\langle E_1[(vector-st! (loc w) n v_3)], #f, s_1 \rangle \rightarrow
\langle E_1[(let ([new (set-marker (o_1 l_1 n v_3))])
  (clear-marker (if (chaperone-of? new v_3)
    (vector-set! l_1 n new)
    (error 'bad-cvref)))]], #f, s_1 \rangle \rightarrow
Either \((o_1 \ l_1 \ n \ v_3)\) reduces to a value or it doesn't (diverges, errors, gets stuck). If the latter, then the same is true for the reduction of \(e_1\). Otherwise, the program state above reduces to

\[
\begin{align*}
&E_1[(\text{let } ([\text{new } o_1 \ l_1 \ n \ v_3]) \\
&\quad (\text{clear-marker} (\text{if } (\text{chaperone-of? } v_3) \\
&\qquad (\text{vector-set! } l_1 \ n \ \text{new}) \\
&\qquad (\text{error} \ '\text{bad-cvref}))))),] \\
&\ #t, s_1>
\end{align*}
\]

if \(v_3\) is not a chaperone of \(v_3\) in \(s_3\), then we get an error. Otherwise the above reduces to

\[
\begin{align*}
&E_1[(\text{let } ([\text{new } v_3'])] \\
&\quad (\text{clear-marker} (\text{if } (\text{chaperone-of? } v_3) \\
&\qquad (\text{vector-set! } l_1 \ n \ v_3') \\
&\qquad (\text{error} \ '\text{bad-cvref}))))], \\
&\ #t, s_3'> \rightarrow \\
&E_1[(\text{if } (\text{chaperone-of? } v_3' \ v_3) \\
&\qquad (\text{vector-set! } l_1 \ n \ v_3') \\
&\qquad (\text{error} \ '\text{bad-cvref}))))], \\
&\ #f, s_3'
\end{align*}
\]

We have that \(\text{chaperone-of}[[s_3', v_3', v_3]]\) and \(s_3' \prec s_1\) (since no inappropriate mutating states are allowed), and the latter via lemma B.8 gives us \(\langle v_3, s_3' \rangle \prec \langle v_4, s_2 \rangle\). Using lemma B.7, that means \(\langle v_3', s_3' \rangle \prec \langle v_4, s_2 \rangle\). Since \(s_3' \prec s_1\), we also have that \(\langle (\text{loc } w), s_1 \rangle \prec \langle (\text{loc } z), s_2 \rangle\) gives us \(\langle (\text{loc } w), s_3' \rangle \prec \langle (\text{loc } z), s_2 \rangle\) via lemma B.7. Since \(\text{loc } w\) points to a chaperone around \(l_1\), we also have \(\langle l_1, s_3' \rangle \prec \langle (\text{loc } z), s_2 \rangle\), which means that \(\langle E_1[(\text{vector-set! } l_1 \ n \ v_3')], \ #f, s_3' \rangle \prec \langle E_2[(\text{vector-set! } (\text{loc } z) \ n \ v_4)], \ #f, s_2 \rangle\)

Thus, we use the IH on the reduction sequence of \(e_2\), the location
corresponding to the chaperoned value (thus removing a single chaperone),
and this approximation to get the rest of the reduction sequence for
e₁, to which we prepend the above steps.

\[
\begin{align*}
\text{e₁} &= E₁[(\text{vector-ref (loc w) n})], \\
&\text{but based on the approximation from} \\
&\text{hypothesis II, there are two possibilities for } s₁(w):
\end{align*}
\]

\[
\begin{align*}
s₁(w) &= (\text{impersonate-vector l₁ m₁ o₁}) \\
\text{Then we get the following reduction step:} \\
\text{E₁}[(\text{vector-ref (loc w) n})], #f, s₁ &→ \\
\text{E₁}[(\text{m₁ l₁ n (vector-ref l₁ n)})], #f, s₁, \\
\text{and } E₁[(\text{m₁ l₁ n (vector-ref l₁ n)})], s₁ &→ \\
\text{E₂}[(\text{m₂ l₂ n (vector-ref l₂ n)})], #f, s₂.
\end{align*}
\]

Applying the IH to the rest of the reduction sequence,
using E₁[(m₁ l₁ n (vector-ref l₁ n))] and s₁ as our new e₁ and s₁
and the approximation above to discharge the hypothesis, we get the rest
of the reduction sequence for our old e₁. As before, we stitch the
reduction step above onto the one (whether divergent, erroring, stuck, or
reduced to a value) we get from the IH.

\[
\begin{align*}
s₁(w) &= (\text{chaperone-vector l₁ m₁ o₁}) \\
\text{Then we get the following reduction step:} \\
\text{E₁}[(\text{vector-ref (loc w) n})], #f, s₁ &→ \\
\text{E₁}[(\text{let ([old (vector-ref l₁ n)])} \\
\text{(let ([new (set-marker (m₁ l₁ n old)])} \\
\text{(clear-marker (if (chaperone-of? new old) \\
\text{new \\
\text{(error 'bad-cvref))}))}), \\
\text{#f, s₁}) \\
\text{Due to the approximation relation, we know that} \\
\text{E₁, s₁} &→ E₂[(\text{loc z}), s₁] (\text{since we skip through chaperones}).
\end{align*}
\]
APPENDIX B. PROOF: CHAPERONE ERASURE

So what we will do is use the entire reduction for $e_6$, but use $E_1[(\text{vector-ref } l_1 n)]$ as $e_1$ (and keep $s_1$ the same), which removes the calculation of a chaperone. From our IH, we get that $<E_1[(\text{vector-ref } l_1 n)], \#f, s_1>$ either:

* Diverges: then the reduction of $e_1$ diverges
* Errors: then the reduction of $e_1$ errors
* Reaches a stuck state: then the reduction of $e_1$ reaches a stuck state.
* Reduces to $<E_1[v_1], \#f, s_3'>$ for some $v_1'$ and $s_3'$

where $<v_1', s_3'> \sim <v_2, s_4>$.

Then in reducing $e_1$, we get the same steps in the RHS of the first let:

$<E_1[(\text{vector-ref } (\text{loc } w) n)], \#f, s_1> \rightarrow$

$<E_1[(\text{let } ([\text{old } v_1'])$

$\text{(let } ([\text{new } (\text{set-marker } (m_1 l_1 n \text{ old})))])$

$\text{(clear-marker } (\text{if } (\text{chaperone-of? } \text{new } \text{ old})$

$\text{new}$

$\text{(error 'bad-cvref))))))],$

$\#f, s_3'> \rightarrow$

$<E_1[(\text{let } ([\text{new } (\text{set-marker } (m_1 l_1 n v_1'))])$

$\text{(clear-marker } (\text{if } (\text{chaperone-of? } \text{new } v_1')$

$\text{new}$

$\text{(error 'bad-cvref))))))],$

$\#f, s_3'> \rightarrow$

$<E_1[(\text{let } ([\text{new } (m_1 l_1 n v_1')])$

$\text{(clear-marker } (\text{if } (\text{chaperone-of? } \text{new } v_1')$

$\text{new}$

$\text{(error 'bad-cvref))))))],$

$\#t, s_3'>$

For reducing $(m_1 l_1 n v_1')$ in this context, there are several cases:

* $<E_1[(\text{let } ([\text{new } (m_1 l_1 n v_1')])$

$\text{(clear-marker } (\text{if } (\text{chaperone-of? } \text{new } v_1')$

$\text{new}$

$\text{(error 'bad-cvref))))]),$
#t, s_3'"> diverges: then reducing e_1 diverges

* \(<\text{E}_1[(\text{let ([new (m_1 l_1 n v_1')])}]
  (\text{clear-marker (if (chaperone-of? new v_1')}\)
  new
  (error 'bad-cvref)))]>,
#t, s_3'> errors: then reducing e_1 errors

* \(<\text{E}_1[(\text{let ([new (m_1 l_1 n v_1')])}]
  (\text{clear-marker (if (chaperone-of? new v_1')}\)
  new
  (error 'bad-cvref)))]>,
#t, s_3'> reaches a stuck state:
then reducing e_1 reaches a stuck state

* \(<\text{E}_1[(\text{let ([new (m_1 l_1 n v_1')])}]
  (\text{clear-marker (if (chaperone-of? new v_1')}\)
  new
  (error 'bad-cvref)))]>,
#t, s_3'> =>

\(<\text{E}_1[(\text{let ([new v_1'])})]
  (\text{clear-marker (if (chaperone-of? new v_1')}\)
  new
  (error 'bad-cvref)))]>,
#t, s_3''> =>

\(<\text{E}_1[(\text{clear-marker (if (chaperone-of? v_1'' v_1')} v_1''))]
  (error 'bad-cvref))],
#t, s_3''> =>

\(<\text{E}_1[(\text{if (chaperone-of? v_1'' v_1') v_1'' (error 'bad-cvref))}]>,
#f, s_3''>

Now there are two cases: v_1'' is not a chaperone of v_1' or it is.

* Not a chaperone: then the reduction of e_1 errors.
APPENDIX B. PROOF: CHAPERONE ERASURE

* Is a chaperone. Then we have chaperone-of[[s_3'', v_1'', v_1']], and

\[ E_1[(if (chaperone-of? v_1'' v_1') v_1'' (error 'bad-cvref)))], #f, s_3'' \]

\[ \rightarrow* \]

\[ E_1[v_1''], #f, s_3'' \].

s_3' <= s_3'', and because of the restrictions on the reduction of e_1, s_3' <-> s_3''. Therefore, \( <v_1', s_3'' > \sim <v_2, s_2> \) by lemma B.8 and by lemma B.7, <v_1'', s_3'' > \sim <v_2, s_2>. Therefore v_1'' is the v_1 we need, and s_3'' is the s_3 we need to finish this case.

\[ E_2[(vector-set! (loc z) n v_4), #f, s_2] \rightarrow \]

\[ E_2[(vector-set! l_2 n (o_2 l_2 n v_4))], #f, s_2 \]

where s_2(z) = (impersonate-vector l_2 m_2 o_2)

e_1 = E_1[(vector-set! (loc w) n v_3)], but based on the approximation from hypothesis II, there are two possibilities for s_1(w):

s_1(w) = (impersonate-vector l_1 m_1 o_1)

Then we get the following reduction step:

\[ E_1[(vector-set! (loc w) n)], #f, s_1 \] \rightarrow

\[ E_1[(vector-set! l_1 n (o_1 l_1 n v_3))], #f, s_1 \]

and \[ E_1[(vector-set! l_1 n (o_1 l_1 n v_3))], #f, s_1 \] \sim

\[ E_2[(vector-set! l_2 n (o_2 l_2 n v_4))], #f, s_2 \]

Applying the IH to the rest of the reduction sequence, using E_1[(vector-set! l_1 n (o_1 l_1 n v_3))] and s_1 as our new e_1 and s_1 and the approximation above to discharge the hypothesis, we get the rest of the reduction sequence for our old e_1. As before, we stitch the reduction step above onto the one (whether divergent, erroring, stuck, or reduced to a value) we get from the IH.

s_1(w) = (chaperone-vector l_1 m_1 o_1)

Then we get the following reduction step:

\[ E_1[(vector-set! (loc w) n v_3)], #f, s_1 \] \rightarrow

\[ E_1[(let ([new (set-marker (o_1 l_1 n v_3))])

(clear-marker (if (chaperone-of? new v_3)\n
...
Either \((o_1 l_1 n v_3)\) reduces to a value or it doesn’t (diverges, errors, gets stuck). If the latter, then the same is true for the reduction of \(e_1\). Otherwise, the program state above reduces to

\[
\begin{align*}
\langle E_1((\text{let } ([\text{new } (o_1 l_1 n v_3)])) \\
\text{(clear-marker if (chaperone-of? new v_3)} \\
\text{(vector-set! l_1 n new)} \\
\text{(error 'bad-cvref))}))], \\
#t, s_1\rangle 
\end{align*}
\]

if \(v_3'\) is not a chaperone of \(v_3\) in \(s_3'\), then we get an error. Otherwise the above reduces to

\[
\begin{align*}
\langle E_1((\text{let } ([\text{new } v_3'])) \\
\text{(clear-marker if (chaperone-of? v_1' v_3)} \\
\text{(vector-set! l_1 n v_3'}) \\
\text{(error 'bad-cvref))}))], \\
#t, s_3'\rangle 
\end{align*}
\]

We have that \(\text{chaperone_of}[[s_3', v_3', v_3]]\) and \(s_3' \leftrightarrow s_1\) (since no inappropriate mutating states are allowed), and the latter via lemma B.8 gives us \(\langle v_3, s_3' \rangle \sim \langle v_4, s_2 \rangle\). Using lemma B.7, that means \(\langle v_3', s_3' \rangle \sim \langle v_4, s_2 \rangle\). Since \(s_3' \leftrightarrow s_1\), we also have that \(\langle \text{loc } w, s_1 \rangle \sim \langle \text{loc } z, s_2 \rangle\) gives us \(\langle \text{loc } w, s_3' \rangle \sim \langle \text{loc } z, s_2 \rangle\) via lemma B.7. Since \(\text{loc } w\) points to a chaperone around \(l_1\), we also have \(\langle l_1, s_3' \rangle \sim \langle \text{loc } z, s_2 \rangle\), which means that
Thus, we use the IH on the reduction sequence of \(e_2\), the location corresponding to the chaperoned value (thus removing a single chaperone), and this approximation to get the rest of the reduction sequence for \(e_1\), to which we prepend the above steps.

\[
\begin{align*}
\langle E_1[(\text{vector-set! } l_1 n v_3'), \#f, s_3'] \rangle & \sim \\
\langle E_2[(\text{vector-set! } (\text{loc } z) n v_4), \#f, s_2] \rangle
\end{align*}
\]

Thus, \(E_2[(\text{vector-set! } (\text{loc } z) n v_4), \#f, s_2] \rangle \sim \langle E_2[(\text{vector-ref } (\text{loc } z) n), \#f, s_2] \rangle \sim \langle E_2[v_4n], \#f, s_2 \rangle \sim \langle E_2[v_4n], #f, s_2 \rangle
\]

where \(s_2(z) = \text{(vector } #f v_4 \ldots v_4n \ldots v_4k)\)

(and \(0 \leq n \leq k\), since \(e_6\) reduces to a value in the context \(E_2\))

\(e_1 = E_1[(\text{vector-ref } (\text{loc } w) n)]\), but based on the approximation from hypothesis II, there are two possibilities for \(s_1(w)\):

\[
\begin{align*}
s_1(w) &= \text{(vector } #f v_3 \ldots v_3n \ldots v_3k)\]
\end{align*}
\]

Then we get the following reduction step:

\[
\begin{align*}
\langle E_1[(\text{vector-ref } (\text{loc } w) n)], \#f, s_1 \rangle & \sim \langle E_1[v_3n], \#f, s_1 \rangle \\
\text{and } \langle E_1[v_3n], s_1 \rangle & \sim \langle E_2[v_4n], \#f, s_2 \rangle, \text{ since the vectors were already approximates in } s_1/s_2. \text{ Thus, } e_5 \text{ reduces to a value (namely, } v_3n)\)
\end{align*}
\]

\[
\begin{align*}
s_1(w) &= \text{(chaperone-vector } l_1 m_1 o_1)\]
\end{align*}
\]

Then we get the following reduction step:

\[
\begin{align*}
\langle E_1[(\text{vector-ref } (\text{loc } w) n)], \#f, s_1 \rangle & \sim \langle E_1[(\text{let } ([\text{old } \text{(vector-ref } l_1 n)]) \\
(\text{let } ([\text{new } \text{(set-marker } m_1 l_1 n \text{ old)}])) \\
(\text{clear-marker } (\text{if } \text{(chaperone-of? } \text{new } \text{old)} \\
\text{new} \\
(\text{error } '\text{bad-cvref}))))]]), \#f, s_1 \rangle
\end{align*}
\]

Due to the approximation relation, we know that

\[
\langle l_1, s_1 \rangle \sim \langle \text{(loc } z), s_1 \rangle \text{ (since we skip through chaperones).}
\]

So what we will do is use the entire reduction for \(e_6\), but use \(E_1[(\text{vector-ref } l_1 n)]\) as \(e_1\) (and keep \(s_1\) the same), which removes
the calculation of a chaperone. From our IH, we get that

\[ \langle E_1[(\text{vector-ref } l_1 n)], \#f, s_1 \rangle \] either:

- Diverges: then the reduction of \( e_1 \) diverges
- Errors: then the reduction of \( e_1 \) errors
- Reaches a stuck state: then the reduction of \( e_1 \) reaches a stuck state.
- Reduces to \( \langle E_1[v_1], \#f, s_3' \rangle \) for some \( v_1' \) and \( s_3' \)
  where \( \langle v_1', s_3' \rangle \sim \langle v_2, s_4 \rangle \).

Then in reducing \( e_1 \), we get the same steps in the RHS of the first let:

\[ \langle E_1[(\text{vector-ref } (\text{loc } w) n)], \#f, s_1 \rangle \rightarrow^* \]

\[
\langle E_1[(\text{let } ([\text{old } v_1'])]
  (\text{let } ([\text{new } (\text{set-marker } m_1 l_1 n \text{ old})])))
  (\text{clear-marker } (\text{if } (\text{chaperone-of? } \text{new } \text{ old})
    \text{new}
    (\text{error 'bad-cvref})))))angle, \#f, s_3' \rightarrow \]

\[
\langle E_1[(\text{let } ([\text{new } (\text{set-marker } m_1 l_1 n \text{ v_1'}))])
  (\text{clear-marker } (\text{if } (\text{chaperone-of? } \text{new } \text{ v_1'}))
    \text{new}
    (\text{error 'bad-cvref}))))rangle, \#f, s_3' \rightarrow \]

\[
\langle E_1[(\text{let } ([\text{new } m_1 l_1 n \text{ v_1'}])]
  (\text{clear-marker } (\text{if } (\text{chaperone-of? } \text{new } \text{ v_1'})
    \text{new}
    (\text{error 'bad-cvref}))))], \#t, s_3' \rangle
\]

For reducing \( m_1 l_1 n \) \( v_1' \) in this context, there are several cases:

- \( \langle E_1[(\text{let } ([\text{new } m_1 l_1 n \text{ v_1'}])]
    (\text{clear-marker } (\text{if } (\text{chaperone-of? } \text{new } \text{ v_1'})
      \text{new}
      (\text{error 'bad-cvref}))))], \#t, s_3' \rangle \) diverges: then reducing \( e_1 \) diverges
APPENDIX B. PROOF: CHAPERONE ERASURE

* $\langle E_1[(\text{let } ([\text{new } (m_1 \ l_1 \ n \ v_1')] )
\text{ (clear-marker (if (chaperone-of? new v_1') }
\text{ new}
\text{ (error 'bad-cvref))))],
\#t, s_3'> \text{ errors: then reducing } e_1 \text{ errors}

* $\langle E_1[(\text{let } ([\text{new } (m_1 \ l_1 \ n \ v_1')] )
\text{ (clear-marker (if (chaperone-of? new v_1') }
\text{ new}
\text{ (error 'bad-cvref))))],
\#t, s_3'> \text{ reaches a stuck state:}
\text{ then reducing } e_1 \text{ reaches a stuck state}

* $\langle E_1[(\text{let } ([\text{new } v_1'']) )
\text{ (clear-marker (if (chaperone-of? new v_1') }
\text{ new}
\text{ (error 'bad-cvref))))],
\#t, s_3'' > \rightarrow$
$\langle E_1[(\text{let } ([\text{new } v_1'']) )
\text{ (clear-marker (if (chaperone-of? new v_1') }
\text{ new}
\text{ (error 'bad-cvref))))],
\#t, s_3'' > \rightarrow$
$\langle E_1[(\text{if (chaperone-of? v_1'' v_1') v_1'' }]
\text{ (error 'bad-cvref))],
\#t, s_3'' > \rightarrow$
$\langle E_1[(\text{if (chaperone-of? v_1'' v_1') v_1'' } (error 'bad-cvref))],
\#f, s_3'' >$

Now there are two cases: $v_1''$ is not a chaperone of $v_1'$ or it is.

* Not a chaperone: then the reduction of $e_1$ errors.

* Is a chaperone. Then we have chaperone-of$[[s_3''', v_1''', v_1']]$,

and
\[E_1[(\text{if} \ (\text{chaperone-of?} \ v_1'' \ v_1') \ v_1'' \ (\text{error} \ 'bad-cvref')))],
\] #f, s_3'' \rightarrow #f, s_3''.

s_3' \leq s_3'', \text{and because of the restrictions on the reduction of e_1, s_3' \sim s_3''}. \text{Therefore,} \langle v_1', s_3'' \rangle \sim \langle v_2, s_2 \rangle \text{by lemma B.8 and by lemma B.7,} \langle v_1'', s_3'' \rangle \sim \langle v_2, s_2 \rangle.

Therefore v_1'' \text{ is the v_1 we need, and s_3''} \text{ is the s_3 we need to finish this case.}

(This exactly mirrors the vector-ref of a chaperoned impersonated vector above, for good reason. I'm not going to repeat it for a chaperoned immutable vector.)

\[E_2[(\text{vector-set!} \ (\text{loc} \ z) \ n \ v_4), \ #f, s_2 \rightarrow]
\] \[E_2[(\text{void}), \ #f, s_2[z \rightarrow (\text{vector} \ #f \ v_4 \ ... \ v_4 \ ... \ v_4k)]]
\text{where} \ s_2(z) = (\text{vector} \ #f \ v_4 \ ... \ v_4n \ ... \ v_4k)
\]
e_1 = E_1[(\text{vector-set!} \ (\text{loc} \ w) \ n \ v_3)], \text{but based on the approximation from hypothesis II, there are two possibilities for s_1(w):}

s_1(w) = (\text{vector} \ #f \ v_30 \ ... \ v_3n \ ... \ v_3k)

Then we get the following reduction step:
\[E_1[(\text{vector-set!} \ (\text{loc} \ w)\ n)], \ #f, s_1 \rightarrow]
\[E_1[(\text{let} \ ([\text{new} \ (\text{set-marker} \ (o_1 \ l_1 \ n \ v_3))])]
and \langle E_1[(\text{void}), \ #f, s_1[w \rightarrow (\text{vector} \ #f \ v_30 \ ... \ v_3 \ ... \ v_3k)]] \sim
\] \[E_2[(\text{void}), \ #f, s_2[z \rightarrow (\text{vector} \ #f \ v_4 \ ... \ v_4 \ ... \ v_4k)]]
\text{(since the only change in the store is replacing the corresponding element in two approximated vectors with approximate values).}

(\text{void}) \text{ is a value, so e_5 evaluates to a value (void) and the resulting expression/store is appropriately approximate to the result of reducing e_6.}

s_1(w) = (\text{chaperone-vector} \ l_1 \ m_1 \ o_1)

Then we get the following reduction step:
\[E_1[(\text{vector-set!} \ (\text{loc} \ w) \ n \ v_3)], \ #f, s_1 \rightarrow]
\[E_1[(\text{let} \ ([\text{new} \ (\text{set-marker} \ (o_1 \ l_1 \ n \ v_3))])]
}
Either \((o_1 l_1 n v_3)\) reduces to a value or it doesn’t (diverges, errors, gets stuck). If the latter, then the same is true for the reduction of \(e_1\). Otherwise, the program state above reduces to

\[
\langle E_1[(\text{let } ([new (o_1 l_1 n v_3)])
\text{(clear-marker (if (chaperone-of? new v_3)
\text{(vector-set! l_1 n new)
\text{(error 'bad-cvref))}))]),
#t, s_1'] \rangle
\]

if \(v_3'\) is not a chaperone of \(v_3\) in \(s_3'\), then we get an error. Otherwise the above reduces to

\[
\langle E_1[(\text{vector-set! l_1 n v_3'}), #f, s_3'] \rangle
\]

We have that \(\text{chaperone-of}[[s_3', v_3', v_3]]\) and \(s_3' \prec \prec s_1\) (since no inappropriate mutating states are allowed), and the latter via lemma B.8 gives us \(\langle v_3, s_3' \rangle \sim \langle v_4, s_2 \rangle\). Using lemma B.7, that means \(\langle v_3', s_3' \rangle \sim \langle v_4, s_2 \rangle\). Since \(s_3' \prec \prec s_1\), we also have that \(\langle \text{loc w}, s_1 \rangle \sim \langle \text{loc z}, s_2 \rangle\) gives us \(\langle \text{loc w}, s_3' \rangle \sim \langle \text{loc z}, s_2 \rangle\) via lemma B.7. Since \(\text{loc w}\) points to a chaperone around
l_1, we also have \( <l_1, s_3'> \sim <(\text{loc } z), s_2>, \) which means that
\[
\begin{align*}
&<E_1[(\text{vector-set! } l_1 n v_3')], #f, s_3'> \sim \\
&<E_2[(\text{vector-set! } (\text{loc } z) n v_4)], #f, s_2>
\end{align*}
\]
Thus, we use the IH on the reduction sequence of \( e_2, \) the location corresponding to the chaperoned value (thus removing a single chaperone), and this approximation to get the rest of the reduction sequence for \( e_1, \) to which we prepend the above steps.

(Again, mirrors the proof of \text{vector-set!} on a chaperoned impersonated vector.)

\[
\begin{align*}
&E_2[(\text{vector-ref } (\text{loc } z) n), #f, s_2] \\
&\rightarrow
\end{align*}
\]
\[
\begin{align*}
&E_2[v_4n], #f, s_2
\end{align*}
\]
where \( s_2(z) = (\text{vector-immutable } v_4 \ldots v_4n \ldots v_4k) \)
(and \( 0 <= n <= k, \) since \( e_6 \) reduces to a value in the context \( E_2)\)

\( e_1 = E_1[(\text{vector-ref } (\text{loc } w) n)], \) but based on the approximation from hypothesis II, there are two possibilities for \( s_1(w): \)

\( s_1(w) = (\text{vector-immutable } v_3 \ldots v_3n \ldots v_3k) \)
Then we get the following reduction step:
\[
\begin{align*}
&E_1[(\text{vector-ref } (\text{loc } w) n)], #f, s_1] \\
&\rightarrow \langle E_1[v_3n], #f, s_1 angle
\end{align*}
\]
and \( \langle E_1[v_3n], s_1 \rangle \sim \langle E_2[v_4n], #f, s_2 \rangle, \) since the vectors were already approximates in \( s_1/s_2. \) Thus, \( e_5 \) reduces to a value (namely, \( v_3n). \)

\( s_1(w) = (\text{chaperone-vector } l_1 m_1 o_1) \)
As before, the proof follows exactly the format of earlier \text{vector-ref}s on chaperoned values, so I’m not repeating it a third time.

**Lemma B.10:**

For all \( e_2 \) that do not contain \text{set-marker}, \text{get-marker}, or \text{chaperone-vector} and \( s_2, \)
\[
\text{Let } E_2[e_6] = e_2.
\]
If there exists no \( v_2 \) or \( s_4 \) such that
\[
\langle E_2[e_6], #f, s_2 \rangle \text{ reduces to } \langle E_2[v_2], #f, s_4 \rangle,
\]
For all \( e_1 \) and \( s_1 \) such that \( \langle e_1, s_1 \rangle \sim \langle e_2, s_2 \rangle \), let \( E_1[e_5] = e_1 \).
Also, require that the reduction of \( \langle e_1, #f, s_1 \rangle \) contains no program states of the form \( \langle E[(\text{vector-set! (loc x) n v}), #t, s \rangle \) where \( s(x) = (\text{vector } #f \text{ v_e ...)}.\)

Either:
1) \( \langle e_1, #f, s_1 \rangle \) diverges
2) there exists a \( b, s_3 \).
   \( \langle e_1, #f, s_1 \rangle \) reduces to \( \langle (\text{error 'variable}), b, s_3 \rangle \)
3) there exists an \( e_3, b, s_3 \).
   \( \langle e_1, #f, s_1 \rangle \) reduces to \( \langle e_3, b, s_3 \rangle \) and \( e_3 \) is a stuck state.

(That is, if the erased program does not reduce the current redex to a value, then the unerased program cannot.)

Proof:

If there's no initial reduction step for \( \langle e_2, #f, s_2 \rangle \), then we have a stuck state, and \( \langle e_1, #f, s_1 \rangle \) will also be a stuck state.
If there is an initial reduction step, then the proof follows the same form as Lemma B.9. Most of the proof just involves stepping in both reduction sequences than inducting, so those stay pretty much the same (that is, we get the same kind of result as the hypothesis, which is that we DON'T reduce to a value). The main difference in this proof is that in the chaperone cases for \( \text{vector-ref/vector-set!}, \text{vector-ref/vector-set!} \) on the chaperoned value (the IH) does _not_ reduce to a value. However, that’s fine, since that's exactly what we want! So in the vector-ref case, this is immediate, since we first vector-ref the chaperoned value. In the vector-set! case, we might either fail to reduce/diverge/error in the function from the chaperone (which is A-OK), or we fail to reduce/diverge/error from doing vector-set! on the chaperoned value.
Theorem 5.4.1:

For all \( e \), if \( e \) is a user-writeable program, \( \text{Eval}(e) = v \), and that evaluation contains no reductions where the left-hand side is of the form 
\( (s \texttt{#t} \texttt{(vector-set!} (\texttt{loc} x) n v_a)) \) where \( s(x) = (\texttt{vector} \ #f \ v_v \ldots) \), then \( \text{Eval}(|e|) = v \).

\(|e|\) is defined as \( e|\text{chaperone-vector}|\rightarrow (\text{lambda} \ (v \ x \ y) \ v)\)

\(e\) is user-writeable means \( e\) contains no uses of set-marker or clear-marker and contains no values of the form \( (\text{loc} \ x) \).

Proof:

Take the reduction sequence for \( <|e|, \ #f, \ {}|\rangle\). Either it diverges, ends in a stuck state, ends in an error state, or ends in a value.

Keep in mind that each reduction step in the erased program has a corresponding reduction step in the unerased programs. (Chaperones only add reduction steps to apply the interceding function and check the returned value for chaperone-ness.)

Diverges:

Ends in a stuck state:

Ends in an error state:

All these cases force the unerased program to NOT reduce to a value as shown in lemma B.10. Therefore these break our initial hypothesis.

Ends in a value:

Let the value state be \( <v_2, \ #f, \ s_2> \). By lemma B.9 and the fact that we know \( <e_1, \ #f, \ {}|\rangle \) reduces to \( <v_1, \ #f, \ s_1|\rangle \) for some state \( s_1 \) (since \( \text{Eval}(e) = v \)), then we know that \( <v_1, \ s_1|\rangle \sim <v_2, \ s_2|\rangle \).

Now let's examine the cases of \( v_2 \):

\( v_2 \) is a boolean: then \( v_1 \) is the same boolean, and \( \text{Eval}(e) = \text{Eval}(|e|) \).

\( v_2 \) is a number: then \( v_1 \) is the same number, and \( \text{Eval}(e) = \text{Eval}(|e|) \).

\( v_2 \) is a pointer to a lambda: then \( v_1 \) must also be a pointer to a...
lambda, and Eval(e) = Eval(|e|) = 'proc'.

v_2 is a pointer to a mutable vector, immutable vector, or impersonator:
Then v_1 is a pointer to the same, or a pointer to a series of chaperones that ends in the same. That is, v_1 cannot contain a lambda. Since Eval only disambiguates locations on whether they contain a lambda or not, and the not case returns 'vector', Eval(e) = Eval(|e|) = 'vector'.
\(<e_1, s> = \approx [e_1, s_1, e_2, s_2, ()]\)

\(\approx [b, s_1, b, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [n, s_1, n, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [\text{void}, s_1, \text{void}, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [\text{chaperone-vector}, s_1, (\lambda x_1 x_2 x_3) x_1, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [\text{chaperone-vector}, s_1, (\text{local} x_1), s_2, ((x y) ...)] =
(\# ((x y) ...))\)

where \((\lambda x_1 x_2 x_3) x_1 = s_2(x_1)\)

\(\approx [\text{prim}, s_1, \text{prim}, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [\text{local} x_1, s_1, e_2, s_2, ((x y) ...)] =
\approx [\text{local} x_1, s_1, e_2, s_2, ((x y) ...)]\)

where \((\text{chaperone-vector} l m o) = s_1(x_1)\)

\(\approx [\text{local} x_1, s_1, (\text{local} x_2), s_2, ((x y) ...)] =
\approx [\text{mutable-vector} l m o, s_1, (\text{mutable-vector} l m o, s_2, ((x y) ...)]\)

where \((\text{immutable-vector} l m o, s_1, (\text{mutable-vector} l m o, s_2, ((x y) ...)]\)

where \((l m o) = s_1(x_1)\)

\(\approx [\text{local} x_1, s_1, (\text{local} x_2), s_2, ((x y) ...)] =
\approx [\text{local} x_1, s_1, (\text{local} x_2), s_2, ((x y) ...)]\)

where \((e_1 = s_1(x_1), e_2 = s_2(x_2)\)

\(\approx [\text{local} x_1, s_1, (\text{lambda} (x_1 ... e_1), s_1, (\text{lambda} (x_2 ... e_2), s_2, ((x y) ...)] =
\approx [\text{local} x_1, s_1, (\text{lambda} (x_2 ... e_2), s_2, ((x y) ...)]\)

\(\approx [(\text{let} ((x_1 x_2 ... e_1), s_1, (\text{let} ((x_1 x_2 ... e_2), s_2, ((x y) ...)] =
\approx [\text{let} ((x_1 x_2 ... e_2), s_1, (\text{let} ((x_1 x_2 ... e_2), s_2, ((x y) ...)]\)

\(\approx [\text{if} e_1 e_2 e_3, s_1, (\text{if} e_1 e_2 e_3, s_2, ((x y) ...)] =
\approx [\text{if} e_1 e_2 e_3, s_1, (\text{if} e_1 e_2 e_3, s_2, ((x y) ...)]\)

\(\approx [\text{error variable}, s_1, (\text{error variable}, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [e, s_1, e_2, s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [(), s_1, (), s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [(e, e_1, ...), s_1, (e, e_2, ...), s_2, ((x y) ...)] =
(\# ((x y) ...))\)

\(\approx [(e, e_1, ...), s_1, (e, e_2, ...), s_2, ((x y) ...)]\)

where \((\# ((x_1 y_1) ...)) = \approx [e_1, s_1, e_2, s_2, ((x y) ...)]\)

\(\approx [(e, e_1, ...), s_1, (e, e_2, ...), s_2, ((x y) ...)] =
(\# ((x_1 y_1) ...))\)

where \((\# ((x_1 y_1) ...)) = \approx [e_1, s_1, e_2, s_2, ((x y) ...)]\)
Bibliography


