Contracts for First-Class Classes

T. Stephen Strickland Matthias Felleisen
PLT @ Northeastern University
{sstrickl,matthias}@ccs.neu.edu

Abstract

First-class classes add expressive power to class-based object-oriented languages. Most importantly, programmers can abstract over common scenarios with first-class classes. When it comes to behavioral software contracts, however, first-class classes pose significant challenges. In this paper, we present the first contract system for a programming language with first-class classes. The design has been implemented for Racket, which supports first-class classes and which implements mixins and traits as syntactic sugar. We expect that our experience also applies to languages with native mixins and/or traits.

Categories and Subject Descriptors D [2]: 3 Object-oriented programming; D [2]: 4 Programming by contract

General Terms Reliability

Keywords contracts, first-class class systems

1. First-class Classes and Contracts

Over the past couple of decades, programming language designers have repeatedly demonstrated the usefulness of operations on classes. For example, Bracha and Cook [7] distilled and formalized the Flavors notion of mixins[1] [39]. Roughly speaking, mixins in a single-inheritance language are functions from classes to classes, adding fields and methods to the given class. Since then, mixins have been the subject of numerous studies [1, 3, 9, 17, 35]. After noticing software engineering problems with mixins, Schärli et al. [44] added traits to Smalltalk. A trait encapsulates only methods. To create a class, a programmer simultaneously derives a class from a superclass and combines it with traits via glue code that manages name clashes or other small mismatches. Both Scala [40] and Fortress [2] support traits.

Smalltalk [20], OCaml [32, 43], and Racket [18], formerly known as PLT Scheme, take the final step and turn classes into first-class values. In such a world, the evaluation of an expression may produce a class. Methods and functions may consume and produce classes. Objects may store classes in fields. And most importantly, the super-class designator in a class is just an expression, meaning the super-class is computed at run-time. Equipped with this power, a programmer can add mixins and traits as "syntactic sugar" [16] but even better, a programmer can explore entirely new ways of organizing class-based programs.

Racket, like Eiffel [38] and many other object-oriented languages [5, 8, 22, 24, 26, 27, 29, 34, 38, 42], also comes with a contract system. A contract [6] is a logical assertion that governs the flow of values between distinct components. The compiler and the run-time system cooperate to monitor contracts so that they can issue an informative report when values fail to respect a contract. In Racket, a report on a contract violator both explains the nature of the violation and pinpoints the violator.

Extending contracts to first-class classes poses significant challenges. First, it demands a language for specifying contracts for classes as values. Second, when classes are first-class values, their separate roles as sources of new objects and as containers for reusable code requires rather different forms of specification than those of Eiffel [38] or JML [30]. Third, implementing contracts for first-class classes affects the performance of classes without contracts. Finally, Racket also mixes conventional inheritance and Beta-style inheritance [21], further complicating the addition of contracts.

Due to these challenges, Racket has not supported contracts for its class system, only for individual objects. Over the past year, we have designed, implemented, and experimented with a contract system that overcomes these obstacles. Furthermore, we have extended the contract system to handle Beta-style inheritance. While our practical work exploits Racket, we conjecture that the ideas apply to other languages with mixins, traits, and similar features.

[1] The concept of mixins first appeared in H. I. Cannon's 1979 paper "Flavors: A non-hierarchical approach to object-oriented programming," a yet unpublished MIT technical report in draft form. Thanks to an anonymous reviewer for pointing to this historical origin.

DLS 2010, October 18, 2010, Reno/Tahoe, Nevada, USA.

2. First-class Classes in Racket

Racket extends the Scheme programming language [45] with a class system, among other additions. In this section, we briefly describe this class system; for details, we refer the interested reader to a tutorial paper [16]. We start with an example program that illustrates the features of the Racket class system. Then we describe the use of classes as first-class values and show how to define mixins, one popular usage pattern. While we do not discuss traits, we encourage the reader to consult Flatt et al.'s report to see traits as syntactic sugar in this world.

2.1 Simple Racket Classes

The core of the Racket class system is similar to that of Java [23]. Figure 1 displays the skeleton of a small module that utilizes some of the basic features. The first line specifies the language of the module. The first two definitions name two new classes: *web%*,[2] a subclass of the root class *object%*, and *logged-web%*, a subclass of *web%*. The body of each class is a mixture of definitions and expressions. The expressions are collected into an initializer, which is evaluated at the end of object construction. During initialization, each class must explicitly invoke the superclass's initializer via the **super-new** expression.

The class *web%* defines *runs-on*, an initialization argument used during object creation that defaults to a value of 8080; a public field *port*, which defaults to the value of *runs-on*; and a private field *listener*, which contains a TCP listener on the given port. Initialization arguments cannot be used in the bodies of methods—they are only valid during the evaluation of expressions that run for object initialization, such as the default values of fields and top-level expressions. In addition, *web%* contains three public methods: *read-request*, which parses an HTTP request from a given input port; *handle-request*, which performs the requested action; and *serve*, which accepts connections on the given port and uses the request parser and handler.

The class *logged-web%* defines a public field *log-file* via the **field** form and a private method *log*. It also uses the **inherit-field** form to gain direct access to the superclass's field *port*. The private method *log* is then used to implement the logging functionality in an overriding declaration of the *handle-request* method. The overridden definition in the superclass is available to the subclass via the **super** form.

The third definition instantiates and then names an object of the *logged-web%* class. This definition provides a different value from the default for the initialization argument *runs-on*. The fourth and last part is a plain expression; it invokes the *serve* method on *server* via **send**.

Naturally, method definitions may refer to **this**, which denotes the object on which the method was invoked. Fields and methods internal to the class are directly accessible via their names, instead of accessing them via **this**. Examples of this convention are the uses of *listener*, *read-request*, and *handle-request* within the method *serve* in *web%*. If the field or method is not defined in the same class, the programmer must declare it as an expected member of the superclass by using the **inherit-field** or **inherit** forms.

[2] The use of **%** at the end of an identifier is a convention used by Racket programmers to signal that a variable stands for a class.

```
#lang racket

(define web%
  (class object%
    (super-new)
    (init [runs-on 8080])
    (field [port runs-on])
    (define listener (tcp-listen port))
    (define/public (read-request in) ...)
    (define/public (handle-request out req) ...)
    (define/public (serve)
      (let-values ([(in out) (tcp-accept listener)])
        (let ([req (read-request in)])
          (close-input-port in)
          (handle-request out req)
          (close-output-port out)
          (serve))))))

(define logged-web%
  (class web%
    (super-new)
    (inherit-field port)
    (field [log-file ...])
    (define/private (log req)
      (call-with-output-file log-file
        (lambda (p)
          (fprintf p "~a: request on port ~a: ~a"
                   (current-seconds) port req))))
    (define/override (handle-request out req)
      (log req) (super handle-request out req))))

(define server (new logged-web% [runs-on 5678]))

(send server serve)
```

Figure 1. A simple object-oriented program in Racket

2.2 First-class Classes and Mixins

Classes in Racket are first-class values. Most importantly, functions may consume and produce classes. The form used to create classes is also an expression like any other in the language. Hence, classes can be created conditionally, or a single class form can be evaluated multiple times, each time resulting in a new class. Thus, the class hierarchy can be arbitrarily extended at run-time.

Figure 2 presents an example of using classes as first-class values. The first function, *add-logging-to-server*, sub-

```
...
;; ASSUME: cls% defines a public method
;; handle-request and a public field port
(define (add-logging-to-server cls%)
  (class cls%
    (super-new)
    (inherit-field port)
    (field [log-file ...])
    (define/private (log req)
      (call-with-output-file log-file
        (lambda (p)
          (fprintf p "~a: request on port ~a: ~a"
                   (current-seconds) port req))))
    (define/override (handle-request out req)
      (log req)
      (super handle-request out req))))

(define (log-and-serve cls%)
  (send (new (add-logging-to-server cls%)) serve))

(log-and-serve web%)

(log-and-serve mail%)
```

Figure 2. Using classes as class values

classes its input class, overriding its *handle-request* method to add logging just as with *logged-web%*. The second takes a class, adds logging via the first function, and then invokes the *serve* method on a new object of the resulting class. The latter function is then used twice with the *web%* class shown earlier and a class *mail%*, a mail server with an interface similar to *web%* whose code is omitted.

The function *add-logging-to-server*, which takes a class that implements the *handle-request* method and extends it with logging behavior, is an example of a *mixin*. Bracha and Cook [7] were the first to propose these kinds of mixins, which provide an abstraction for extending classes with the same new behavior which is especially useful in a single inheritance language. As seen here, mixins are naturally expressed in a language with first-class classes.

3. Contracts

Behavioral software contracts provide a notation for describing the expected behavior and use of values via obligations and promises. We start with a description of the contract system of Eiffel [38], with which Meyer first popularized the idea of design by contract in the object-oriented world. We then switch to the contract system in Racket and elaborate on those attributes that shape our design of contracts for first-class classes: contract boundaries, blame tracking, and contracts for higher-order values.

3.1 Contracts in Eiffel

Eiffel supports three ways of describing contracts on classes: 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.

```
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
```

Figure 3. An Eiffel class for a stack of prime numbers

Figure 3 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.

Since all class invariants are checked on both method entry and exit, this can lead to problems due to re-entrance [49, 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# [4], provide the programmer with a mechanism to state that a block of code is allowed to break invariants. After the block is executed, the invariants are checked to ensure that they are properly re-established. Racket solves this problem in a different way, as we explain next.

```
#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 (compose not empty?) . →d . prime?)]
 [add-and-remove (number? . → . number?)]
 [map-prime ((prime? . → . prime?) . → . void?)])
```

Figure 4. A Racket module for a stack of prime numbers

3.2 Contracts in Racket

To illustrate contracts in Racket, figure 4 contains a module, written in the plain imperative fragment of the language with a "translation" of the Eiffel class plus two additional procedures. The contracts are written using a syntactic extension of Racket that allows prefix operators to be written in an infix manner. We use this extension to move $\rightarrow$ and $\rightarrow_{\mathbf{d}}$, the contract operators for procedures, between the domain and range contracts. 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 *prime-stack* module and uses this functionality, the argument to *push* and the result from *pop* are checked for primality. Expressions that are checked as preconditions can be specified in the $\rightarrow_{\mathbf{d}}$ contract form via the #:pre keyword, as in the contract of *pop*. Thus, the stack must be non-empty when *pop* is called.

3.2.1 Contract Boundaries

In contrast to Eiffel, 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. Calls internal to the module are *not* checked. Consider the nonsensical *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 *prime-stack*, these calls are not checked and therefore do not signal a contract violation.

At first glance, contract checking at boundaries is too coarse grained to be useful. However, it eliminates the above-mentioned re-entrance problem and ensures that contract checking doesn't interfere with the implementation of tail calls, a critical element of proper object-oriented design. Furthermore, the experience of the Racket community pragmatically demonstrates that boundary-checked contracts are highly useful. Programmers tend to write small modules and trust the code within a single module. When contract violations are signaled, the resulting error limits the region of code that needs to be inspected. Programmers debug via unit tests and, if they fail to find the precise location of the error, employ **define/contract** and **with-contract** [47]. These create local contract boundaries that can narrow down the search within a module.

3.2.2 Blame Tracking

Every module that imports *prime-stack* enters into an agreement with the *prime-stack* module. If a contract violation occurs, the appropriate party to the contracts 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 in type systems.

3.2.3 Contracts for Higher-Order Values

Since Racket is a functional language, functions are first-class values. Of course, it is impossible to check contracts, such as (*prime? . → . prime?*), on functions 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 contracts on the return value(s) hold after the function is evaluated.

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 to the server and then the result from the server 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, and thus the positive party 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 each time we descend into the left-hand side of a function contract. For details, see Findler and Felleisen's paper [15].

4. Adding Contracts to First-class Classes

While contract systems for object-oriented languages have been around for a long time, none of them cope with first-class classes. The presence of first-class classes poses significant challenges to the introduction of contracts. In this section, we spell out the two linguistic problems; section 6 deals with the performance problem. First, a language with first-class classes must come with contracts that are separate from the classes that they protect. Second, when entire classes flow across contract boundaries, the contract sublanguage must separate their roles as factories of new objects and as containers of reusable code.

4.1 Separating Classes and Contracts

In conventional object-oriented languages with contracts, such as Eiffel or Spec#, programmers write down contracts as part of a class or method. Writing down contracts next to fields and methods feels natural to programmers who know only fixed class hierarchies. In a language with first-class classes, however, contracts must come separate from classes. After all, classes can flow across contract boundaries just like numbers, strings, or objects.

The flow of classes across contract boundaries can take all shapes and forms. Consider a class-consuming method (or function). Like its analog that demands a prime number, this method should be able to require that its input—a class—meet certain conditions. Similarly, a class-producing method (or function) should be able to make promises about its results. Now if we think of mixins as "functions" that extend unknown superclasses, it becomes obvious that class contracts are the natural way to specify their promises and obligations. Similarly, trait manipulating operations that alias some methods, remove others, and modify a third class can be described with such class contracts, too.

4.2 The Role of Classes

Classes play two major roles in an object-oriented language: they serve as blueprints for the creation of new objects, and they serve as containers of code. While the former role is well-explored from a contract perspective, the latter role demands a new look because classes are no longer in a fixed relationship. Indeed, one and the same class may extend many different superclasses, even though the language supports only single inheritance. Consequently, we must decide whether a subclass is required to be consistent with the superclass's behavior [33] or whether it exploits the superclass solely as a code repository.

As far as contracts are concerned, the separate roles of classes pose two design problems. First, we need to decide whether a class contract should specify both roles jointly or whether a contract may specify the two different aspects separately. Second, we must also decide whether contracts protect implementations or interfaces.

For guidance, we inspected the Racket code base (about 500 Kloc). This inspection suggests that the creators of classes wish to impose different obligations and expect different promises from subclasses and from objects. Our contracts therefore allow the specification of obligations on subclasses, similar to but also different from so-called specialization interfaces [28]. Furthermore, the code inspection also implies that many uses of class hierarchies are about implementation inheritance rather than type inheritance.[3] The design of our class contracts therefore focuses on protecting implementations, not behavioral type relationships.

5. A Tour of Contracts for First-class Classes

Our main construct is a contract *combinator* that combines *clauses*. Each clause is an assertion on visible fields and methods. Following the preceding section, the clauses come in two flavors: one for external method calls and one for the specialization interface. The latter govern the flow of values between methods in superclasses and subclasses.

5.1 A Contract Combinator for Class Values

Class contracts are specified with the **class/c** combinator. Like the contract combinator for first-class functions, **class/c** exhibits a higher-order flavor. After all, a class denotes a collection of objects, i.e., values with behavior, and it is impossible to check behavioral aspects all at once. For example, contracts for methods are checked like contracts for functions—one application at a time. Contracts on initialization arguments provide a second example, as contract application must be delayed until the actual instantiation is performed. Similarly, our contract system applies the contract to the value in a field whenever the field is accessed or mutated from across a contract boundary.

Figure 5 contains the equivalent of the *PRIME_STACK* class from figure 3. In the **class/c** contract, each method is paired with a function-like contract that specifies the expected values for all arguments, starting with **this**, followed

[3] It is likely that Racket's untyped nature biases our results here and that a typed language has different needs.

by a description of the range. These descriptions can either be predicates or contracts. The **any/c** contract is a trivial contract that never causes a contract error. To appreciate the contract for **this**, take a look at the contract for *pop*, which demands non-empty stacks.

```
#lang racket

(define pstack%
  (class object%
    (super-new)
    (define intlist null)
    (define/public (empty?) (null? intlist))
    (define/public (push n) (set! intlist (cons n intlist)))
    (define/public (pop)
      (begin0 (car intlist)
              (set! intlist (cdr intlist))))))

(define (non-empty-stack? o) (not (send o empty?)))

(provide/contract
 [pstack%
  (class/c [empty? (any/c . → . boolean?)]
           [push (any/c prime? . → . void?)]
           [pop  (non-empty-stack? . → . prime?)])])
```

Figure 5. A Racket class for a stack of prime numbers

Figure 6 illustrates how the separation of contract specifications from the class definitions helps us deal with first-class classes. Specifically, assume *web* is the name of the module from figure 1, and that it exports the class *web%*. This class is imported into two distinct modules, each of which re-exports the class with a contract. Here, we show only the contract on the initialization argument *runs-on*. The module at the top checks for super-user privileges and allows any valid port number to be used; in contrast, the module at the bottom assumes an unprivileged user and therefore enforces that the programmer provides a port number larger than 1024 when instantiating the class.

Finally, the separation of contracts from classes also renders contracts for features such as mixins straightforward. Since mixins are essentially functions from superclasses to subclasses, the contract on a mixin can be described as a function contract whose domain is constrained by one class contract and whose range is described by a second one. The domain contract is applied to the argument at the time the mixin function is applied, and the range contract is applied to the resulting class value. See figure 7 for a sample contract for the code from figure 2.

5.2 Contracts for Instantiation vs. Extension

Figure 7 also highlights an important distinction between different uses of class contracts. The domain contract of the function is *not* a specification of how objects instantiated

```
#lang racket
(require web)
(check-superuser-access!)
...
(provide/contract
 [web% (class/c (init [runs-on valid-port?]) ...)])
```

```
#lang racket
(require web)
(define (high-port? n)
  (and (valid-port? n) (>= n 1024)))
(provide/contract
 [web% (class/c (init [runs-on high-port?]) ...)])
```

Figure 6. Exporting a class with different contracts

from the class are used but rather what features are expected from the viewpoint of a subclass. Instead of describing how fields and methods of objects created from the class should behave, as in the range contract, it describes the expected behavior of inherited fields and **super** calls. The contract also contains an **override** clause for *handle-request*. Specifically the clause demands that the given class defines (or overrides) *handle-request* and that all method calls to *handle-request* in the superclass satisfy the specified method contract.

```
#lang racket

(define (add-logging-to-server cls%)
  (class cls%
    (super-new)
    (inherit-field port)
    (field [log-file ...])
    (define/private (log req)
      (call-with-output-file log-file
        (lambda (p)
          (fprintf p "~a: request on port ~a: ~a"
                   (current-seconds) port req))))
    (define/override (handle-request out req)
      (log req) (super handle-request out req))))

(define request-handler/c
  (any/c output-port? valid-request? . → . void?))

(provide/contract
 [add-logging-to-server
  (→ (class/c (inherit-field [port tcp-port?])
              (override [handle-request request-handler/c])
              (super [handle-request request-handler/c]))
     (class/c (field [log-file valid-path?])
              [handle-request request-handler/c]))])
```

Figure 7. A contract for a mixin

Following section 4.2, **class/c** contracts support seven different kinds of clauses. Each protects a class from uses via instantiated objects or conventional inheritance:

1. untagged clauses protect uses of methods via objects;
2. **field** clauses protect uses of fields via objects;
3. **init** clauses describe contracts for object creation;
4. **inherit** clauses impose obligations on the use of methods by subclasses;
5. **inherit-field** clauses impose obligations on the use of fields by subclasses;
6. **super** clauses impose obligations on subclasses for uses of the superclass's implementation; and
7. **override** clauses document that a method is overrideable and its expected interface.

The functionality of Eiffel-like invariants can be macro-expressed by a layer of syntactic sugar. The new contract form takes the expected invariants, along with the other contract features listed above, and creates a use of **class/c** where each method contract includes the invariants as preconditions and postconditions.

5.3 Contracts and Objects

While a class contract specifies the behavior of every object of a protected class, in some cases a programmer may wish to place a special contract on some object. One abstract example is the Singleton pattern [19]. Since only one instance is provided to clients, the class itself never crosses the contract boundary. Hence, the contract must be attached to the object as it flows out of the module.

Since an object has already been fully initialized and it doesn't play the role of a code container in a class-based language, only external field and method clauses appear in object contracts. Uncontracted and contracted uses of the object must share the same state, which was not the case for the previously existing object contracts in Racket. If the client retrieves the value from a field that is inconsistent with that field's contract, then the server is blamed.

A programmer may also wish to enforce a stronger contract on a particular object than on its class. Take the prime stack implementation in figure 5. In the same module, we may wish to create an object from that class and ensure that it is used only with prime numbers greater than 1,000,000. Figure 8 contains an appropriate code snippet.

5.4 Contracts and Class Inspection

Racket, like other object-oriented languages, comes with operations for inspecting the relationships between objects and classes and among classes. Specifically, **is-a?** determines whether some object is an instance of some class, and **subclass?** checks whether a given class is a subclass of another one. This, in turn, raises the question whether an object or

```
...
(define (large-num? n)
  (and (number? n) (> n 1000000)))

(define big-prime-stack (new pstack%))

(provide/contract
 [big-prime-stack
  (object/c [empty? (any/c . → . boolean?)]
            [push (any/c large-num? . → . void?)]
            [pop  (non-empty-stack? . → . large-num?)])])
```

Figure 8. A prime stack object with a stronger contract

class extension satisfies the same relationships with a contracted version of the class.

Figure 9 contains a program with multiple contract boundaries; a pictorial view of the same program is given in figure 10. In the latter, vertical lines represent contract boundaries, squares are classes, and circles denote instances. The program contains three modules, and each exports an object and a single contracted class. Here, we must decide whether the instances of the uncontracted classes still have an **is-a?** relationship to the exported contracted versions.

r1

```
#lang racket

(define cls1% (class object% ...))
(define cls2% (class cls1% ...))
(define o1 (new cls2% ...))

(provide/contract [cls2% c1] [o1 any/c])
```

r2

```
#lang racket
(require r1)
(is-a? o1 cls2%)

(define o2 (new cls2% ...))
(define cls3%
  (class cls2% ...))
(define o3 (new cls3% ...)))

(provide/contract
  [cls3% c2] [o3 any/c])
```

r3

```
#lang racket

(require r2)

(is-a? o3 cls3%)

(define o4 (new cls3%)

...

...
```

Figure 9. Example uses of **is-a?**

For our design decision, we accept the principle that the removal of contracts should not affect the observable outcome if the program with contracts evaluates to a plain value. The theory of contracts as projections [13] explains this principle from a theoretical perspective. A projection is roughly speaking a function that maps values to values with "less" behavior or here, values where some part of

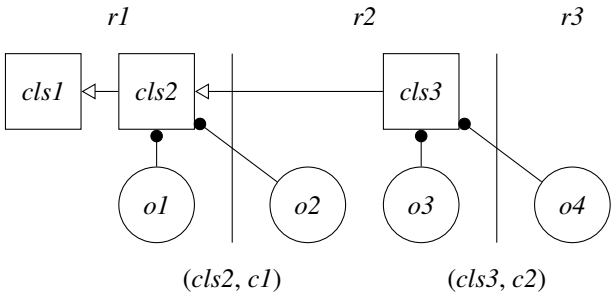

Figure 10. Classes, objects, and contract boundaries

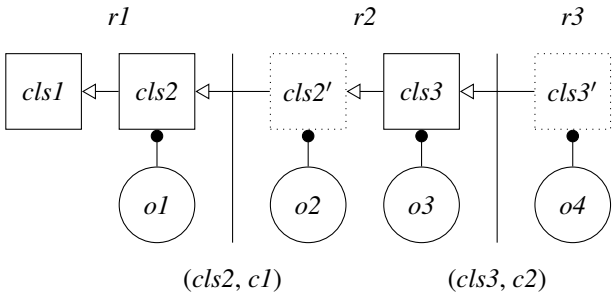

Figure 11. New classes after applying contract projections

the behavior has been replaced by errors. In the case of contract projections, the act of applying a projection turns inappropriate behavior into an error signal that represents a contract violation.

When we apply this principle to the **is-a?** and **subclass?** checks, the consistent answer is that a class with a contract should act as much as possible as its counterpart without contracts. Since, in the absence of contracts, **is-a?** and **subclass?** would return true for objects and subclasses of a given class, they must also return true when those same objects and subclasses are compared against the contracted version of the class. Thus, all the queries in figure 9 return true, because they would do so in a version of the program where none of the class definitions were contracted.

6. Implementation

The Findler-Blume explanation of contracts as pairs of mathematical projections also motivates an implementation of a contract monitoring system. For a language that uniformly views values as first-class entities, the implementation of a contract is almost a transliteration of the mathematical model into functional code.

Thus, **class/c** denotes a function that maps the (names of the) two contract parties into a pair of projections from classes to classes. One way to understand this idea is to compare figures 10 and 11. In the latter 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.

6.1 Basics

Both classes and objects are implemented as heterogeneous and opaque structures.[4] 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 an auxiliary function formed from the initialization expressions of the class (see section 2.1).

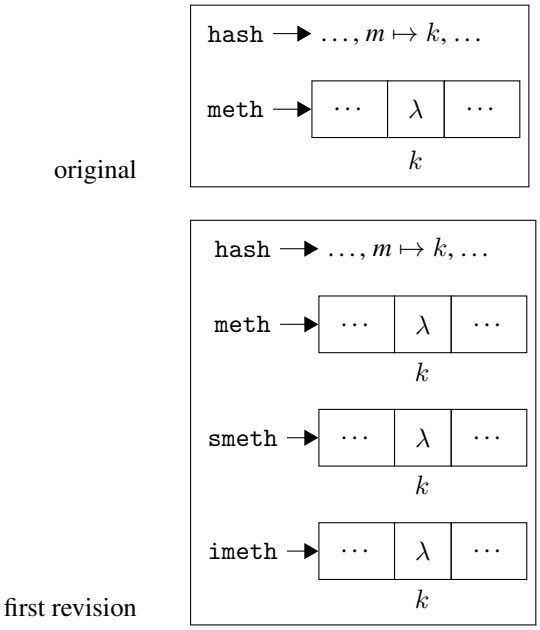

Figure 12. Method table organization

As usual, the class representation contains a vector of methods and a hash table that maps a method name into an index for the vector. Fields are represented in the same fashion, except that instantiated objects contain the vector of field values. The class compiler transforms both a field access and a method call into computations that use the appropriate vector slot. The compiler can compute the vector index for internal uses of fields and methods statically, whereas the index must be computed at runtime for external references. Field assignments are converted into vector assignments. Calls to **super** methods are compiled via indexing computations into the method table of the super class.

In short, all method calls access the same method vector, whether they are internal calls, external **send**s, or **super** calls in a subclass. Each kind of contract clause affects a different set of uses, however. We could store the different projections and apply the correct projection for a particular

[4] Racket's class system is technically implemented via a macro library, though it is programmatically indistinguishable from a native class system.

use, but applying contract projections can be a costly operation. Our implementation therefore trades space in the class representation for run-time speed; that is, it separates out the method vector into three vectors, illustrated by figure 12. External calls via **send** use the `meth` vector, **super** calls access `smeth`, and direct internal calls access `imeth`. In support, the revised compiler converts each operation into an indexed access into the appropriate vector. This separation immediately supports eager projection application for the methods accessed via **send** and super calls; section 6.2 describes the additional changes required for internal calls.

For mutable fields, we cannot apply contract projections upon object creation. Since the value of a field changes over time, we apply appropriate projections whenever a field is accessed or mutated across a contract boundary. To this end, our compiler stores two pairs of functions for each field. One pair protects external uses, and the other pair protects internal uses. Each pair contains an accessor and a mutator function. When a contract with a field-related clause is applied to a class, the appropriate pair of functions is replaced with a new pair. The new accessor function applies the appropriate projection with covariant blame to the result of the old accessor. The new mutator function applies the projection with contravariant blame to the incoming value.

To protect initialization arguments, the compiler creates a contracted initialization function. It consumes the same arguments as the original one, applies the appropriate projection to each argument, and passes the contracted arguments to the original initialization function. The only trickiness is due to keyword initialization of arguments, where multiple classes in a single hierarchy can have initialization arguments of the same name. These overlapping names are resolved in order from subclass to superclass, affecting the order in which contracts are specified. Of course, a Java-like initialization method does not pose this problem.

6.2 Internal Dynamic Dispatch

The revised object organization cannot support contracts on dynamic dispatches within the class hierarchy. Such method calls should be affected only by 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. Also, since we need to ensure that **override** contracts are not lost when a method is overridden, each class also contains an analogous table of vectors of projections. 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. The resulting changes to the class structure are illustrated in figure 13.

Figure 14 contains an example class hierarchy where one of the classes, *cls2*, is contracted with **inherit** and **override** clauses for the method *m*. Figure 15 depicts the method and

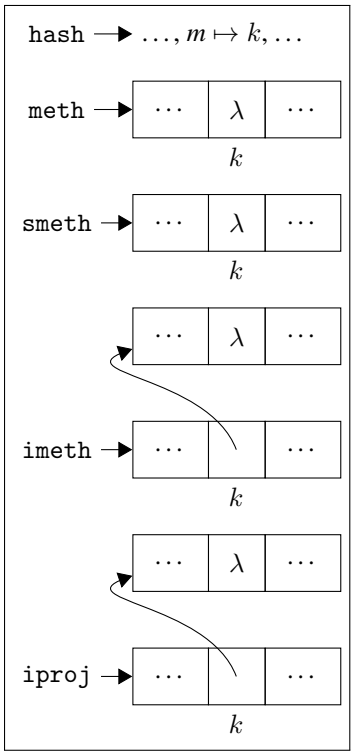

Figure 13. Second revision of class method tables

m1

```
#lang racket

(define cls1
  (class object% ... (define/public (m x) ...)
         (define/public (n) (m ...))))

(define cls2 (class cls1 ... (define/override (m x) ...)))

(provide/contract
 [cls2 (class/c ... (inherit [m c_i]) (override [m c_o]))])
```

m2

```
#lang racket

(require m1)

(define cls3
  (class cls2 ... (inherit m) (define/public (p) (m ...))))

(define cls4
  (class cls3 ... (define/override (m x) ...)
         (define/public (q) (m ...))))
```

Figure 14. Example classes with **override** and **inherit** contracts

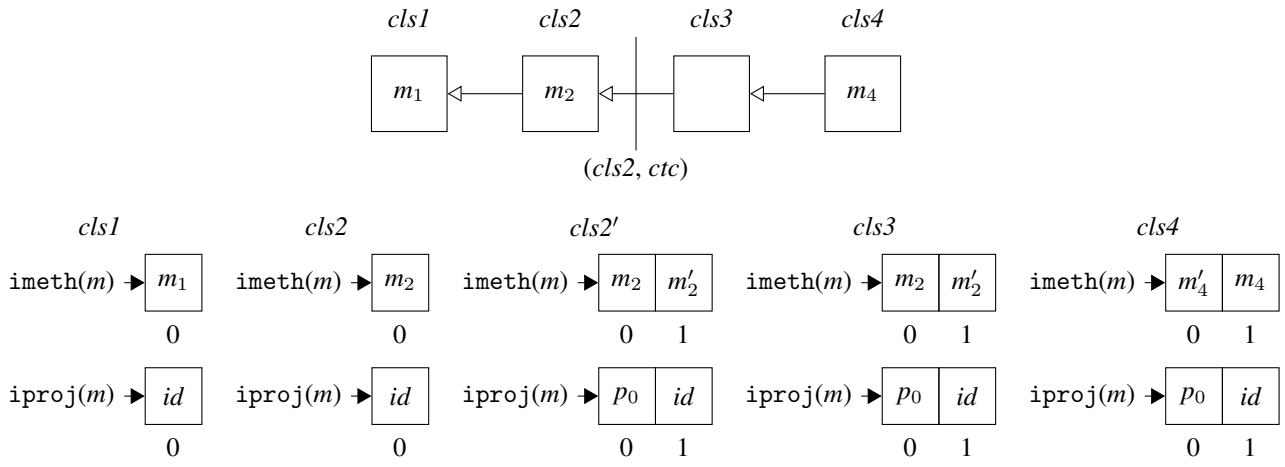

Figure 15. Dispatch tables for figure 14

projection vectors used for internal calls to *m* in each class. Class *cls2* is shown twice: *cls2* denotes the class without contracts, and *cls2′* depicts the class with contracts applied. The different method implementations are differentiated using subscripts, and primed method names denote application of contract projections to 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, *cls1* and *cls2* both use a one-element vector for method *m* since there are no contract boundaries between them and *cls1*, the first class to define *m*. In contrast, the contract region introduced by **provide/contract** separates *cls3* and *cls4* from *cls2*, meaning the two classes use a vector of two elements for method *m*.

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 *cls4*. If code using that instance calls *n*, the call from *cls1* crosses the contract boundary to use the overridden method in *cls4*. Since the call in *cls1* is compiled to retrieve the method from index 0, it uses the contracted method m'_4. If the code calls *q*, the internal method call from *cls4* uses index 1. Thus, it retrieves m_4, i.e., the uncontracted method.

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. See the diagram for method *m* in *cls1*.

If a subclass does not override a method, as in *cls3*, its method and projection vectors are the same as its superclass. If it does override the method, as in *cls2* and *cls4*, 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 *cls4*, the method at index 0 is m'_4, the result of composing m_4 with the projection p_o. The method at index 1 is just m_4, since index 1 of the projection vector still contains the identity projection.

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 overridding subclass.

The creation of the method and projection vectors for *cls2′* from those in *cls2* follows these rules. The method in index 0 has been copied, while the method m'_2 at index 1 is the result of applying the contract projection for c_i (from the class contract for *cl2*) to m_2. The projection in index 0 is the result of composing the new projection for c_o to the identity projection, and the projection at index 1 is the identity.

6.3 Object Contracts

Since classes in Racket are values, the contract combinator for objects can be described in terms of the contract combinator for classes. Operating on an object with a contract—where the contract specifies the behavior of public methods and fields on one particular object—is equivalent to operating on an object that shares the same state but is based on a class equipped with the appropriate contracts.

When an object flows across a contract boundary that applies an object contract, our run-time system creates a new object whose methods are proxies for the original methods and whose field selectors and mutators forward actions to the original object. This forwarding guarantees that the two objects—the original and the one with the object contract—share state as desired. Figure 16 explains the idea pictori-

	rev	avg	min	max	norm
External field	pre	2657	2564	2744	1.00
access	class/c	2597	2499	2914	0.98
(2.5×10^6)	object/c	2749	2648	2813	1.03
External method	pre	3303	3253	3380	1.00
access	class/c	3320	3257	3398	1.01
(2.5×10^6)	object/c	3306	3249	3354	1.00
Local field	pre	2089	2087	2091	1.00
access	class/c	2090	2087	2103	1.00
(2.5×10^7)	object/c	7602	7461	8287	3.64
Inherited field	pre	2305	2303	2306	1.00
access	class/c	2305	2303	2306	1.00
(2.5×10^7)	object/c	7779	7667	8447	3.37
Local method	pre	1659	1655	1679	1.00
access	class/c	1743	1727	1774	1.05
(2.5×10^6)	object/c	1878	1840	1947	1.13
Inherited method	pre	1662	1650	1687	1.00
access	class/c	1741	1729	1773	1.05
(2.5×10^6)	object/c	1873	1835	1900	1.13

	rev	avg	min	max	norm
Instantiation	pre	6756	6569	7610	1.00
(2.5×10^5)	class/c	6576	6514	6618	0.97
	object/c	7276	6989	8482	1.08
Subclassing	pre	4220	4149	4333	1.00
(2.5×10^5)	class/c	4373	4296	4442	1.04
	object/c	4790	4663	4955	1.14
Subclassing	pre	4461	4387	4567	1.00
(new fields)	class/c	4879	4772	4971	1.09
(2.5×10^5)	object/c	5272	5123	5382	1.18
Subclassing	pre	4566	4471	4692	1.00
(new methods)	class/c	4867	4782	4965	1.07
(2.5×10^5)	object/c	5235	5072	5453	1.15
Subclassing	pre	4710	4646	4821	1.00
(fields/methods)	class/c	5323	5226	5442	1.13
(2.5×10^5)	object/c	5775	5615	5965	1.23

All benchmarks were executed ten times on a Intel i7-860 (2.8GHz) with 8 GB of DDR3 1333 memory running Ubuntu 9.10.

Figure 17. Micro-benchmark results (times in milliseconds)

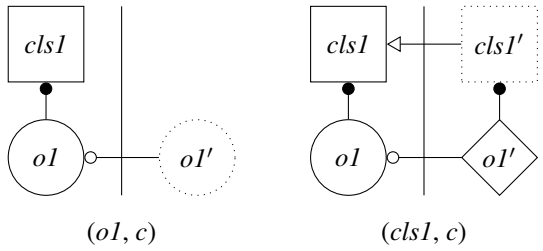

Figure 16. Example application of an object contract

ally, using a diamond on the right-hand side of the contract boundary to represent a proxied object. The diagram also points out that this latter object contains a reference to the contracted class for method access and a reference to the original object for fields.

To ensure that **is-a?** and **subclass?** return the same result on wrapped classes and/or objects, the class system must determine that two classes are equal modulo contract wrapping. For this, each class contains a reference to the uncontracted version of the class. In unwrapped classes, this is a self-reference. Otherwise, the new contracted class copies the reference from the original class during contract application. Then the checks in the implementations of **is-a?** and **subclass?** use reference equality on the original classes.

7. Evaluation

The addition of linguistic constructs to a language demands two forms of evaluation. On the one hand, designers and programmers need to know whether a new construct imposes any additional costs on the other constructs of the language. On the other hand, designers also need to demonstrate how useful the new construct is for the existing code base. In this section, we report on our evaluation of contracts for first-class classes with respect to these two criteria. The first subsection focuses on performance, the second on the utility of our new contracts.

7.1 Performance Evaluation

Our implementation strategy affects the performance of all operations on both classes and objects. Worse, even classes and objects that do not use contracts pay a price. The latter effect is mostly noticeable in the context of micro-benchmarks that stress-test various features of the class system. These micro-benchmarks compare three versions of the Racket code base: the initial code base, dubbed "pre;" the "class/c" code base, which runs all existing code using the new class compiler; and the "object/c" code base, in which object contracts are implemented via the revised class compiler. Since none of the benchmarks use class contracts, the difference in running times describe precisely the added overhead of the new implementations.

The results of our micro-benchmarks are displayed in figure 17. All times are given in milliseconds, and for each test we provide the average running time, the minimum running time, the maximum running time, and the average running time normalized to the "pre" version. We also list how often the benchmark executes the measured feature. Notice that the local and inherited field benchmarks employ an order of magnitude more trials than the first three, and the instantiation and inheritance tests an order of magnitude fewer. The four benchmarks for subclass creation differ in whether new fields and/or methods are introduced.

When only class contracts are introduced, the overhead on even these class-focused benchmarks is quite small—usually less than 5%. In a couple of cases, the average running time actually drops because our revised class compiler can better optimize field accesses due to the separation of internal and external accesses. Once object contracts and thus object proxying are added, however, the overhead increases noticeably in all cases but external accesses.

For internal field access—a fairly common use case—the impact of proxying is huge. While the original compiler translates field references into direct structure dereferences, the revised compiler must take into account that some objects are proxy objects. Hence all internal field references have slowed down by approximately 350%.[5]

Fortunately micro-benchmarks tend to exaggerate the effect of our compiler changes. In a realistic code base, internal field references aren't used for every other instruction, and methods perform real work rather than return the same numeric value for every call. Therefore we also measure the effects on the addition of contracts to the class system with two macro-benchmarks. See figure 18 for the results.

	rev	avg	min	max	norm
universe	pre	7834	7735	7903	1.00
teachpack	class/c	7767	7716	7856	0.99
program	object/c	7758	7678	7808	0.99
DrRacket	pre	494608	491790	497510	1.00
interaction	class/c	495133	493220	498070	1.00
test	object/c	518873	517680	520370	1.05

All benchmarks were executed ten times on a Intel i7-860 (2.8GHz) with 8 GB of DDR3 1333 memory running Ubuntu 9.10.

Figure 18. Macro-benchmark results (times in ms)

The first macro-benchmark is a program written using the "universe" library, which provides students with the tools to write purely functional distributed, interactive programs [11]. The library is written using Racket's class system. The program that measures the effect of the class system changes on the "universe" teachpack strongly exercises the features provided by the teachpack. The benchmark shows no additional overhead on such programs.

The second macro-benchmark is an automated interaction test for DrRacket [14], formerly DrScheme. DrRacket is a Racket-specific graphical IDE whose implementation makes heavy use of classes and objects. The automated interaction test of the IDE allows the team to measure its performance. The team tracks these measurements during development of the code base to ensure that no change unduly slows down DrRacket. Figure 18 shows that the addition of class contracts causes no noticeable impact. Indeed, even the overhead of object proxying is a tolerable 5%.

In summary, the addition of contracts to our language of first-class classes has some impact on all class-based programs, not just those that employ contracts. While micro-benchmarks suggest that this cost is significant for some features of the class system, the load for complex systems is almost too small to be noticed.

7.2 Feature Evaluation

Although the presence of first-class classes naturally leads to a class contract such as **class/c**, the question arises whether all of its expressive power is truly needed. At this point we lack definitive pragmatic evidence, but we do have some preliminary anecdotal insight that we wish to share.

To evaluate the expressiveness of **class/c**, we equipped several heavily used core classes of the DrRacket IDE with lightweight contracts.[6] The generated contracts assume that external method calls and calls to overridden methods must satisfy the same contracts. Starting up DrRacket, however, immediately reveals that these contracts fail because subclasses make different assumptions than object clients.

One particularly revealing example concerns the so-called *snip%* class, which specifies that two methods, *get-extent* and *get-text*, consume both mandatory and optional arguments. Several subclasses break this assumption. They override the methods so that all arguments are mandatory. Even though these subclasses are not behavioral subtypes of *snip%*, they are safe because *snip%* internally dispatches to these methods with all arguments provided explicitly.

Without allowing for contracts that separate object interfaces from specialization interfaces, we would need to revisit every subclass of *snip%*, of which there are over 20 inside the DrRacket code base alone. For each of them, we would need to change the implementation of *get-extent* and *get-text*. Instead, we change the contract so that overriding classes only need to support the full-arity version of those methods. For external uses of these methods, we stick to the original, flexible contracts.

The separation of method and field contracts into external and internal contracts also allows the programmer to encode policy for field and method access. Figure 19 contains a mixin similar to existing mixins in the Racket code base. The *add-locking* mixin assumes a superclass that contains a public method *run*. When invoked on a suitable superclass, *add-locking* returns a subclass that executes calls to run in a sequential manner. It also provides another method, *run/nolock*, that can be used by subclasses to execute the service without locking. Only subclasses are expected to use this method, as well as the *lock* and *unlock* methods, but Racket's class system does not currently provide the equivalent of Java's **protected** keyword.

Since our contracts allow the programmer to separately describe how external and internal method calls should be protected, we can use contracts to mimic protected methods. We show part of the resulting contract in figure 19. The ex-

[5] We are currently investigating an alternate implementation strategy for objects, which may eliminate a portion of this overhead.

[6] We actually used a tool by Klein and Findler [25] to generate contracts from documentation and adjusted the results manually.

```
(define (add-locking cls%)
  (class cls%
    (super-new)
    (define mutex (make-mutex))
    (define/override (run cmd)
      (mutex-lock mutex)
      (super run cmd)
      (mutex-unlock mutex))
    (define/public (run/nolock cmd) (super run cmd))
    (define/public (locked?) (mutex-locked? mutex))
    (define/public (lock) (mutex-lock mutex))
    (define/public (unlock) (mutex-unlock mutex))
    ...))

(define (locked? o) (send o locked?))

(provide/contract
 [add-locking
  (→ (class/c ...
       (super [run (locked? any/c . → . any/c)]))
     (class/c ...
       [run (any/c any/c . → . any/c)]
       (inherit [run/nolock (locked? any/c . → any/c)]
       [run/nolock (none/c none/c . → . any/c)])))])
```

Figure 19. A mixin to add locking and its contract

ternal contract for *run/nolock* ensures that the method cannot be invoked on objects by using **none/c** for the arguments. The **none/c** contract always fails, and here the resulting failure blames the party that called the method. The internal contract, specified by **inherit**, checks that the mutex has already been locked prior to being called. This ensures that the superclass's *run* method is not accidentally invoked during other invocations of that method. Similarly, the contract on the mixin's input checks all uses of the superclass's *run* method within any subclasses.

8. Adding Beta-style Augmentation

In conventional class-based languages, subclasses may override methods. Beta [31] and gbeta [10] instead demand that subclasses provide *method augmentations*, which the super method may utilize. This shift of perspective favors behavioral subtyping; the superclass dictates the primary behavior of a method and how it may change.

Goldberg et al. [21] describe how to mix conventional inheritance and Beta-style inheritance in a single language. Mixing the two styles requires the programmer to specify two attributes for each method definition: whether a method overrides or augments the superclass's method and whether a method can be augmented or overridden in a subclass. Overriding methods can be used only if the previous definition of the method specified that the method is overrideable, and an augmenting method can be used only if the previous definition is augmentable.

Naturally, mixing the two styles in one language complicates the semantics of dynamic dispatch. On initial entry, the *first* augmentable method declaration is executed. If no augmentable method is found, then the *last* overriding method declaration is executed. When a superclass wishes to use a subclass's augmentation, specified by **inner**, the same search starts at the current class's subclass, if any. If no subclass augments the method, a default expression is evaluated instead. If a method overrides a previous definition, then the superclass's implementation can be accessed via **super**.

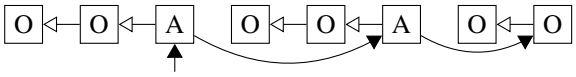

Figure 20. Mixed inheritance chains

Figure 20 illustrates this mix of inheritance for a single method. Each class is labeled with "A" if the method is defined to be augmentable and "O" if it is overrideable. The solid arrow entering the first augmentable method is the initial entry point. The other solid arrows denote methods executed via **inner**, and the arrows with outline heads point to the method that are executed via **super**.

Racket supports both Beta-style inheritance and its use alongside conventional inheritance. Thus, our contract system can protect augmentable methods as well as overrideable methods. Guarding calls via internal dynamic dispatch to augmentable methods is simpler than for overrideable methods. The first augmentable implementation of a method, when available, is the target of dynamic dispatch for that class and any future subclasses. Thus, the projections for contracts that guard against internal dynamic dispatch to augmentable methods can be applied eagerly. Only methods that can be overridden require projections to be stored.

Augmentations of a augmentable method accessed by **inner** can be declared overrideable, however. Thus, to protect **inner** calls, the class implementation contains a projection for each method that is applied when an augmentation is added or overridden. The projection is reset to the identity projection when the new augmentation is augmentable, which adds another link to the chain of **inner** calls. An accompanying technical report [48] more fully describes the extensions in support of Beta-style inheritance.

9. Related and Future Work

While Parnas [41] introduced the idea of using declarative specifications of expected behavior between different software components, Meyer [36, 37] named them contracts and, with his design of Eiffel [38], put them on the map for software engineers and language designers. Later work by Carrillo-Castellon et al. [8] and Plösch [42] added similar contract systems to dynamically typed object-oriented

languages. Findler and Felleisen [12] revisited Meyer's contracts with a focus on blame tracking. Contracts for higher-order functions are also due to Findler and Felleisen [15].

The contracts presented in this paper protect classes consistent with implementation inheritance. Our work thus calls for two future extensions: interface inheritance and declaration of contracts inside class definitions.

First, Racket, like other object-oriented languages, enables the programmer to specify interfaces and attach them to classes. These interfaces describe public methods that a class must implement. Class definitions can then list implemented interfaces. Our plan calls for contracts in interfaces. Classes that implement a contracted interface will have their methods checked according to behavioral subtyping [12].

Second, we would like to allow programmers to declare contracts on class features inside a class definition, instead of contracting the resulting value. This implies, however, that as in our previous work on contracts for first-class modules [46], the class itself is a contract boundary. Doing so requires further research into how class-level contract boundaries interact with each other and other contract boundaries.

Acknowledgments The research benefited from advice by Matthew Flatt and Robby Findler. Jan Vitek's comments on an early draft helped us improve the presentation.

References

[1] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 96–114, 2003.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, 2008.

[3] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.

[4] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[5] Detlef Bartetzko, Clemens Fischer, Michael Moller, and Heike Wehrheim. Jass—Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[6] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, June 1999.

[7] Gilad Bracha and William Cook. Mixin-based inheritance. In *Object-Oriented Programming, Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990.

[8] M. Carrillo-Castellon, J. Garcia-Molina, E. Pimentel, and I. Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 7(9):23–28, 1996.

[9] Dominic Duggan and Ching-Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 223–240, 2001.

[10] Erik Ernst. *gbeta—a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, University of Aarhus, Århus, Denmark, 1999.

[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional I/O system, or fun for freshman kids. In *International Conference on Functional Programming*, pages 47–58, September 2009.

[12] R. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–15, 2001.

[13] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Functional and Logic Programming*, volume 3945 of *LNCS*, pages 226–241. Springer, April 2006.

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

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

[16] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, pages 270–289, November 2006.

[17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[18] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `http://racket-lang.org/tr1/`.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[20] Adele Goldberg and David Robinson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[21] David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. Super and inner—together at last! In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 116–129, 2004.

[22] Benedict Gomes, David Stoutamire, Boris Vaysman, and Holger Klawitter. *A Language Manual for Sather 1.1*, August 1996.

[23] J. Gosling, B. Joy, and G. L.Steele. *The Java Language Specification*. Addison-Welsley, 1996.

[24] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of Meta-Level Architectures and Reflection*, volume 1616 of *LNCS*, pages 175–196, July 1999.

[25] Casey Klein, Matthew Flatt, and Robert Bruce Findler. Random testing for higher-order, stateful programs. In *Object-*

Oriented Programming, Systems, Languages, and Applications, 2010. To appear.

[26] Michael Kölling and John Rosenberg. *Blue: Language Specification, version 0.94*, 1997.

[27] Reto Kramer. iContract: The Java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, page 295, 1998.

[28] John Lamping. Typing the specialization interface. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–214, September 1993.

[29] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Not.*, 12(2):1–79, February 1977.

[30] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[31] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[32] Xavier Leroy. *The Objective Caml system, Documentation and User's guide*, 1997.

[33] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[34] D. C. Luckham and F.W. von Henke. An overview of Anna, a specification language for Ada. In *IEEE Software*, volume 2, pages 9–23, March 1985.

[35] Sean McDirmid, Matthew Flatt, and Wilson Hsieh. Jiazzi: new-age components for old-fashioned Java. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222, 2001.

[36] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[37] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[38] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[39] David A. Moon. Object-oriented programming with flavors. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–8, 1986.

[40] Martin Odersky. *The Scala Language Specification*. Ecole Polytechnique Fédérale de Lausanne, 2009.

[41] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[42] R. Plösch. Design by contract for Python. In *IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference*, pages 213–219, December 1997.

[43] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Symposium on Principles of Programming Languages*, pages 40–53, January 1997.

[44] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer, July 2003.

[45] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009.

[46] T. Stephen Strickland and Matthias Felleisen. Contracts for first-class modules. In *Symposium on Dynamic Languages*, pages 27–38, October 2009.

[47] T. Stephen Strickland and Matthias Felleisen. Nested and dynamic contract boundaries. In *Implementation and Application of Functional Languages*, September 2009. To appear.

[48] T. Stephen Strickland and Matthias Felleisen. Contracts for first-class classes. Technical Report NU-CCIS-10-04, Northeastern University, 2010.

[49] Clemens Szyperski. *Component Software*. Addison-Wesley, 1997.