A

Contracts for First-Class Classes

T. STEPHEN STRICKLAND, CHRISTOS DIMOULAS, ASUMU TAKIKAWA, and MATTHIAS FELLEISEN, Northeastern University, Boston, MA 02115

First-class classes enable programmers to abstract over patterns in the class hierarchy and to experiment with new forms of object-oriented programming such as mixins and traits. This increase in expressive power calls for tools to control the complexity of the software architecture. A contract system is one possible tool that has seen much use in object-oriented programming languages, but existing contract systems cannot cope with first-class classes. On the one hand, the typical contract language deals only with plain values such as numbers, while classes are higher-order values. On the other hand, contract specifications are usually contained within class definitions, while classes as values call for a separate contract language.

This paper presents the design and implementation of a contract system for first-class classes as well as a two-pronged evaluation. The first one states and proves a "blame correctness" theorem for a model of our language. The theorem shows that when the contract system assigns blame to a component for a contract violation, the component is indeed responsible for providing the non-conforming value. The second part, consisting of benchmarks and case studies, demonstrates the need for the rich contract language and validates that our implementation approach is performant with respect to time.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms: Reliability

Additional Key Words and Phrases: Contracts, first-class class systems

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 57 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. FIRST-CLASS CLASSES AND CONTRACTS

First-class classes enable the programmer to dynamically pick context-appropriate base classes, to load new classes at run-time to implement a plug-in architecture, or to describe generic extensions to class hierarchies. With first-class classes, programmers can experiment with new styles of program design. For example, mixins [Moon 1986; Bracha and Cook 1990] and traits [Schärli et al. 2003; Fisher and Reppy 2004] turn into programmer-defined constructs. Many dynamically typed object-oriented languages, such as Python, Racket [Flatt and PLT 2010], Ruby [Flanagan and Matsumoto 2008], and Smalltalk [Goldberg and Robinson 1983], support classes as first-class values in some form. Research projects, such as MetaFJig [Servetto and Zucca 2010],

This research was conducted at Northeastern University and was supported in part by the US Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, and and the National Science Foundation.—The authors' current addresses are: (1) Strickland, Department of Computer Science, University of Maryland, A.V. Williams 115, College Park, MD 20742; (2) Dimoulas, School of Engineering and Applied Sciences, Harvard University, Maxwell-Dworkin 309, Cambridge, MA 02138; (3) Takikawa and Felleisen, College of Computer Science, Northeastern University, 202 West Village H, Boston, MA 02115.

DOI: http://dx.doi.org/10.1145/0000000.0000000

attempt to inject this same expressive power into languages like Java where classes exist primarily as static entities.

Designs for mixins come in many flavors [Allen et al. 2003; Ancona et al. 2003; Duggan and Techaubol 2001; Flatt et al. 1998; McDirmid et al. 2001]. In Racket, mixins are simply functions from classes to classes. Traits are an alternative that address software engineering weaknesses found in mixin-based systems. Instead of abstracting over a superclass, traits bundle class features that the programmer combines in various ways with both other traits and existing classes. Fortress [Allen et al. 2008] comes with built-in traits, and languages with first-class classes can easily provide them as syntactic sugar [Flatt et al. 2006].

Given the flexibility of first-class classes, we need tools to ensure that the expectations between the various components of a program remain expressible and enforceable. Because first-class classes often appear in untyped languages, we have chosen to explore run-time contracts. Most contract systems for object-oriented languages [Bartetzko et al. 2001; Carrillo-Castellón et al. 1996; Gomes et al. 1996; Karaorman et al. 1999; Kölling and Rosenberg 1997; Kramer 1998; Luckham and von Henke 1985; Lampson et al. 1977; Plösch 1997] follow the tradition pioneered by Bertrand Meyer [Meyer 1992b]. Classes contain annotations that describe pre- and postconditions for methods as well as invariants on fields. Each annotation is a boolean expression that triggers a contract violation if it evaluates to false. The method preconditions and class invariants are evaluated on method entry, and if the contract system detects a contract violation, the caller is blamed. The method postconditions and class invariants are run on method exit, and the contract system blames the method if a contract violation occurs.

These existing contract systems can express only conditions that are checked immediately, as shown in section 2. Put differently, these systems cannot directly express [Felleisen 1991] conditions about values with behavior, e.g., objects, classes, or closures; instead such contracts require encodings and code duplication. In addition, the contracts must be specified as part of the classes themselves. Two classes that come with the same contract cannot refer to a separately-defined contract to make the sharing explicit. Each class must reproduce the entire contract text as appropriate annotations in its body.

In fact, because objects are also values with behavior and state, existing contract systems cannot adequately protect objects, much less first-class classes. They provide no direct way to describe the expected behavior of an object in terms of its behavior instead of its class's position in the class hierarchy. This is true even of contract systems for dynamic languages like Ruby or Python, where "duck typing," or depending on provided behavior instead of an object's pedigree, is predominant. Adding first-class classes to the mix only compounds the problem, because programmers need to specify the behavior they expect from arbitrary superclasses and behaviors that they provide as class extensions. The need for such specifications conflicts with the tight binding of contracts to class forms.

In addition, Eiffel-like contract systems assume that subclasses are behavioral subtypes [Liskov and Wing 1994] of their superclass. However, programmers often treat classes simply as modules of reusable code. In doing so, classes that import or inherit from a given class may exhibit different behavior than their superclasses. In languages such as Ruby, this is taken to the extreme where classes are just modules that may also be instantiated to create objects. Thus, the Eiffel-based assumption that subclasses behave like their superclass may restrict the applicability of the contract system to existing programs.

Our proposed contract system overcomes all these shortcomings. To do this, we take inspiration from Findler and Felleisen [2002]. Their contracts for higher-order behav-

ioral values introduce delayed checks. Instead of tightly coupling contracts and the values they represent, separately-defined contracts are "wrapped" around values. The programmer also specifies *contract boundaries* between program components, which are the points where values are wrapped with contracts. In other words, the crossing of values across contract boundaries determines what contracts are in effect. The fact that classes are open, extensible values where extension may take place across these contract boundaries raises interesting questions of which contract boundaries should be in effect during dynamic dispatch.

In this paper,[1] we adapt the contract system of Findler and Felleisen to first-class classes. We present a design together with a comprehensive evaluation. Via an operational model, we prove that class contracts assign blame correctly if a contract is violated [Dimoulas et al. 2011]. Via the annotation of a large code base, we examine the utility of the proposed design. We benchmark the newly contracted code to determine whether the proposed implementation of class contracts is suitable for real-world use. Since our implementation of class contracts requires changes to the existing class system, we also provide benchmarks to evaluate the impact on the class system for non-contracted code.

Section 2 contrasts the Eiffel contract system and the Findler and Felleisen contract system as implemented in the Racket programming language [Flatt and PLT 2010]. Section 3 examines the Racket class system [Flatt et al. 2006] and presents a design for contracts that protect first-class classes. Section 4 explains the design as a formal model, section 5 extends the model with additional annotations needed to prove blame correctness, and section 6 uses the extended model to prove that class contracts assign blame accurately. Section 7 spells out our strategy for implementing contracts for first-class classes. Section 8 describes the impact of our implementation on existing code as well as an approach to adding contracts to an existing class-heavy code base to measure the performance and utility of the new contracts. We discuss related work in section 9.

2. CONTRACTS

Behavioral software contracts provide a notation for describing the expected behavior of components via obligations and promises. We start with a description of Eiffel's contracts [Meyer 1992b], with which Meyer first popularized the idea of design by contract in the object-oriented world. We then switch to the Racket contract system and elaborate on the attributes that shape our design of contracts for first-class classes: 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 1 sketches the class *PRIME_STACK*, which implements a stack of prime numbers. It assumes a predicate *is_prime* on integers and a class *LIST* that implements a mutable list with *is_empty*, *add* and *remove* operations. Both *add* and *remove* operate on the same end of the list. The class contains a private field *intlist*, in

[1] This paper extends a preliminary version [Strickland and Felleisen 2010a] with a theoretical model, soundness theorem, and a comprehensive evaluation.

```
class PRIME_STACK
create
    make
feature
    is_empty : BOOLEAN is
        do
            Result := intlist.is_empty
        end
    push(new_item : INTEGER) is
        require
            item_is_prime: is_prime(new_item)
        do
            intlist.add(new_item)
        end
    pop : INTEGER is
        require
            stack_is_nonempty : not is_empty
        do
            Result := intlist.remove
        ensure
            is_prime(Result)
        end
feature {NONE}
    make
        do
            create intlist
        end
    intlist : LIST [ INTEGER ]
invariant
    intlist_not_void : intlist /= Void
end
```

Fig. 1. An Eiffel class for a stack of prime numbers

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 constructor for the *PRIME_STACK* class, *make*, initializes the field to an newly created empty list. The class provides three public operations: *push*, *pop*, and *is_empty*. 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.

Figure 2 contains an extension to *PRIME_STACK* that adds a stack transformation function *map*. The *map* method takes a transformation function from prime numbers to prime numbers, applies this function to each item in the internally stored list, stores the results of these applications in a new list, and finally replaces the internal list with the new one. To encode the requirement on the function argument, we create a new abstract class *PRIME_TRANSFORMER* that contains an *apply* method. The argument and result of any concrete implementation of the *apply* method are checked as being prime numbers.

While this example demonstrates that it is in principle possible to encode Findler-Felleisen-style higher-order contracts in Eiffel, doing so imposes a heavy burden on the software engineer. First, it requires the creation of a new class for each higher-order

```
class PRIME_STACK
feature
    ...
    map(f : PRIME_TRANSFORMER) is
        local
            new : LIST [ INTEGER ]
        do
            create newlist
            from intlist.start until intlist.after loop
                newlist.extend(f.apply(intlist.item))
                intlist.forth
            end
            intlist := newlist
        end
    ...
end

deferred class PRIME_TRANSFORMER
feature
    apply(arg : INTEGER) is
        require
            arg_contains_prime : isPrime(arg)
        deferred
        ensure
            res_is_prime : isPrime(Result)
        end
end
```

Fig. 2. Extending *PRIME_STACK* to include stack transformation

check. This example needs a new class for the functional argument to *map* to ensure that an integer-to-integer function maps primes to primes. Integer-to-integer functions mapping positives numbers to positives would need another separate class, and so on for each possible constraint on the domain or range of such functions. Second, because of Eiffel's nominal type system, the programmers of a large system would have to agree on a *central* repository for all such classes to ensure the largest possible reuse. Finally, the example also poses an Eiffel-specific problem. Ideally, such functions should take advantage of Eiffel's **agent** construct for creating functions and procedures from existing methods. Doing so is impossible, however, because we would not be able to impose the desired constraints, which require a true subclass. With structural contracts à la Findler and Felleisen, all of these issues disappear.

Another problem with Eiffel contracts concerns the checking of class invariants. They are checked on both method entry and exit, which leads to problems due to re-entrance [Szyperski 1997, p.66]. Any method that needs to temporarily break an invariant must perform all the actions necessary to restore the invariant locally, call the desired method, and undo its actions.

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. Spec#'s mechanism to relax contract checking is orthogonal from *where* contracts are placed.

In contrast, Racket relaxes contract checking when values do not cross any contract boundaries.

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

(define empty-stack null)
(define (empty? s) (null? s))
(define (push s i) (cons i s))
(define (pop s) (values (car s) (cdr s)))
(define (map-prime s f) (map f s))
(define (stack/c elem/c) (listof elem/c))
```

```
#lang racket   ;; name: stack

(require stack-impl)

(provide/contract
 [empty-stack (stack/c any/c)]
 [empty? (→ (stack/c any/c) boolean?)]
 [push (→ (stack/c any/c) any/c (stack/c any/c))]
 [pop (→ (and/c (stack/c any/c) (λ (s) (not (empty? s))))
         (values any/c (stack/c any/c)))]
 [map-prime (→ (stack/c any/c) (→ any/c any/c) (stack/c any/c))])
```

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

(require stack-impl)

(define (prime? n) ...)

(provide/contract
 [empty-stack (stack/c prime?)]
 [empty? (→ (stack/c prime?) boolean?)]
 [push (→ (stack/c prime?) any/c (stack/c prime?))]
 [pop (→ (and/c (stack/c prime?) (λ (s) (not (empty? s))))
         (values prime? (stack/c prime?)))]
 [map-prime (→ (stack/c prime?) (→ prime? prime?) (stack/c prime?))])
```

Fig. 3. Racket modules for a stack implementation

2.2. Contracts in Racket

Figure 3 illustrates the use of Findler and Felleisen's contracts in Racket. It contains code that implements the above Eiffel class as a stateless, functional module, dubbed *stack-impl*. Here, each operation also takes the stack on which to operate, and the *pop* function must return both the popped value and the changed stack. The *stack/c* contract combinator is also provided. This combinator takes a contract that describes the elements in the stack and returns a contract that can be applied to appropriate stacks. In addition, there are two modules that re-export the implementation with different contracts on the implementation's behavior. The first module, dubbed *stack*, exports a stack that contain any type of value. The second one, dubbed *prime-stack*,

contains a predicate on numbers called *prime?* and checks that the values in the stack adhere to that predicate.

The prefix $\rightarrow$ operator creates contracts for functions. The initial arguments to $\rightarrow$ describe the arguments to the function, and the final argument describes the result. The **values** form can be used within the last argument to $\rightarrow$ to describe multiple return values. Both modules provide *push* and *pop* operations on the stack. In addition to the expected contracts, we also check that the stack argument to *pop* is both the expected type of stack and that the stack is non-empty. To do this, we use the **and/c** contract combinator, which applies multiple contracts to the same value. To check for a non-empty stack, we write an ordinary Racket predicate, to illustrate that any predicate in Racket is a valid contract.

2.2.1. Contract Boundaries. Contracts in Racket are checked only when values cross a contract boundary—in this case, a module boundary. For our running example, a boundary crossing occurs when a value flows into or out of the *stack* or *prime-stack* module via function calls. Since the contracts are only checked on these module boundaries and do not alter the original implementation, this allows us to export the common stack implementation with two different sets of contracts without duplicating code.

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 popular design patterns. Finally, boundary-oriented contracts eliminate the above-mentioned re-entrance problem in a pragmatic manner.

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*.

Our notion of blame pinpoints whether a party to the contract—the server or the client—violates its contractual obligations. Upon a violation detection, the blame information narrows down the search space for a fix to just two possible sources:

— the blamed component, for the case that the contract describes a desirable property of the component or,
— the contract itself, for the case that the expectations that the contract imposes are undesirably strict for the blamed component.

In the first case, the programmer needs to fix the blamed component to uphold the contract. In the latter case, the programmer needs to revise the contract to match the desired expectations.

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 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 ($\rightarrow$ *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. Wrapper objects are the natural way to implement this form of checking. The key is to equip the wrapper with enough information so that it 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 then the result 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. For details, see the work by Findler and Felleisen [2002].

3. CONTRACTS FOR FIRST-CLASS CLASSES

Racket includes a class system [Flatt et al. 2006] loosely related to those found in languages such as 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 by considering several classes from a hypothetical windowing library inspired by Racket's GUI toolkit. This sample class definition shows a class for modal dialogs:

```
(define dialog%
  (class window%
    (super-new)
    (init-field label)
    (field [min-width 300]
           [min-height 300])
    (init-field [width min-width]
                [height min-height])
    (define/public (undersize? [w width] [h height])
      (or (< w min-width) (< h min-height)))
    (define/override (draw-widget)
      (super draw-widget)
      (draw-label label))
    (define/private (draw-label label)
      ...)))
```

The first expression in a **class** form specifies the superclass. In this case, the superclass is a built-in class *window%* of the GUI library.[2] The call to **super-new** initializes the superclass. The **field** clause contains the names of other public fields and their initialization expressions. The **init-field** clause lists expected initialization arguments,

[2]Class names in Racket are suffixed with % by convention

here a label, initial width, and initial height of the dialog, and also creates public fields for the provided values. The width and height initialization arguments are optional and are set to the minimum width and height fields by default. The keywords **define/public**, **define/override**, and **define/private** respectively designate a public method, an overriding method, and a private 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. For example, the *draw-label* private method is called from *draw-widget* as if it were a local function. Within an overriding method form, the **super** keyword is used to call the corresponding superclass method. In *dialog%*, the *draw-widget* method relies on the *window%* widget's drawing method.

Method definitions support features such as optional and keyword-based arguments. For example, the *undersize?* method takes two optional arguments, *w* and *h*, that are respectively set to the values of the *width* and *height* fields by default.

Here is a subclass of *dialog%* that represents a dialog for opening files:

```
(define path-dialog%
  (class dialog%
    (super-new)
    (init-field path)
    (inherit-field label)
    (define/public (get-path) path)
    (define/public (set-path! new-path)
      (set! path new-path))
    (define/public (get-file-size)
      (file-size path))
    (define/override (draw-widget)
      (super draw-widget)
      (draw-path path))
    (define/private (draw-path)
      ...)))
```

The **inherit-field** clauses require the superclass to have the listed fields and allow the subclass to use those fields as if they were defined locally.

The *path-dialog%* class overrides the definition of *draw-widget* to draw a new label that displays the current path shown in the dialog. Like in C#, a Racket class must explicitly override a public method. In Racket, the superclass may not be statically determinable; the override specification helps the class system check at runtime that the superclass contains this public method when *path-dialog%* 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:

```
(define (make-resizable base-class)
  (class base-class
    (super-new)
    (inherit undersize)
    (inherit-field width height)
    (define/public (resize w h)
      (set! width w)
      (set! height h))
    (define/public (halve)
      (resize (/ width 2) (* height 2)))))
```

```
(define resizable-dialog (make-resizable dialog%))
(define resizable-path-dialog (make-resizable path-dialog%))
```

The second definition applies the *make-resizable* mixin from the first definition to the *dialog%* class, and the third one uses it on *path-dialog%*. Like the **inherit-field** clauses introduced earlier, an **inherit** clause ensures the superclass contains the listed methods and allows the subclass to use those methods as if they were defined locally.

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

```
(define d1 (new dialog% [label "D1"] [width 400] [height 500]))
(define d2 (new path-dialog% [label "D2"] [path "/usr/bin/racket"]))
(define d3 (new (make-resizable dialog%) [label "D3"] [width 400]))
```

Names and values for initialization arguments are paired. Recall that the *width* and *height* initialization arguments are optional, and they are omitted in several cases. Here, we create a regular dialog *d1* and a path dialog *d2*. To illustrate the flexibility of **new**, we also create an object from the result of mixing the *resize* method into the *dialog%* class and label it *d3*.

In addition to conventional inheritance, Racket also provides 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.

3.1. Contracts on Classes

Let us illustrate the language for class contracts with a contract for *dialog%*:

```
(define dialog%/c
  (class/c
    (init-field [label (string-len/c 256)])
    (field [min-weight (≥/c 0)] [min-height (≥/c 0)])
    (init-field [width (≥/c 0)] [height (≥/c 0)])
    [draw-widget (m→ #:pre (not (send this undersize?)) void?)]
    [undersize? (m*→ () ((≥/c 0) (≥/c 0)) boolean?)]))
```

The $\overset{m}{\rightarrow}$ and $\overset{m^*}{\rightarrow}$ contract combinators are the method equivalents of the $\rightarrow$ and $\rightarrow$* contract combinators. The $\rightarrow$* contract combinator has two sequences of domain contracts: one for required arguments and one for optional arguments.

According to this contract, a dialog has three important features. First, a dialog has labels that are provided at instantiation. Second, a dialog has a pre-set minimum width and height. Third, a dialog can be drawn and report whether it is undersized.

In general, the **class/c** combinator takes a series of (tagged) clauses of names and contracts. Contract clauses without a tag, such as those for *draw-widget* or *undersize?*, describe how object clients may use methods. 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. Methods may also have preconditions, which use **this** to access the target object. For example, calls to *draw-widget* via **send** must occur only on dialogs that are not undersized.

One important point to note here concerns the intent of our class contracts. In an untyped world, classes tend to serve as vessels of reusable code. Hence inheritance

just mimics module import, as programmers expect in many dynamic languages, and classes are free to act differently than their superclasses. Consequently, our class contracts check the behavior of the protected class, but place no restrictions on overridden behavior in subclasses. In particular, if a subclass later extends a contracted dialog by overriding the *draw-widget* method, the contract is not enforced on the new overriding implementation. If control flow reaches the implementation in the contracted dialog via a call from its subclass, however, it is checked appropriately.

Class contracts can be part of a function contract:

```
(define make-resizable/c
  (→ (class/c (inherit-field [width (≥/c 0)] [height (≥/c 0)])
              (inherit [undersize? (→* () ((≥/c 0) (≥/c 0)) boolean?)]))
     (class/c [resize (→dm [w (≥/c 0)] [h (≥/c 0)]
                           #:pre (not (send this undersize? w h))
                           void?)]
              (override [resize (→m (≥/c 0) (≥/c 0) void?)]))))
```

Here, the domain contract specifies that the superclass must contain the public method *undersize?* and fields *width* and *height*. It also contains contracts that check uses of these methods and fields in the subclass. The range contract includes an ordinary method specification. As this contract illustrates, a class contract differentiates the object interface from the specification interface [Lamping 1993]. That is, the contract may impose distinct constraints on the behavior of object clients and subclasses.

At first glance, a programmer might question the practicality of contracts for mixins. After all, contracts could simply be applied to the concrete classes that result from a mixin application. However, mixins are used in situations where truly dynamic class creation is required. Consider this function header for a function *open-file* that queries the user for a file via a dialog:

```
(define (open-file message dialog-mixin) ...)
```

The *open-file* function takes a message and a mixin that allows the client programmer to customize the query dialog (via subclassing) before it is shown to the user. Since *open-file* is provided by an opaque library, the client programmer needs assurance that methods inherited from the base class provided by *open-file* behave correctly. Clauses like **inherit-field** from the domain of the mixin contract from earlier ensure this. In short, the mixin contract can protect the mixin from unknown parent classes.

Both **inherit** and **override** clauses deal with dynamic dispatch inside the class hierarchy, but they specify different clients and servers for a given method implementation. Contracts in **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 **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, **override** clauses are particularly useful in the context of the template-and-hook pattern.

If class contracts allowed only access to listed class features, then the range contract above would restrict clients of the subclass to one method: *resize*. Instead, class contracts are *translucent*, i.e., they allow unlisted class features to be used or extended without constraints. For example, the *undersize?* and *draw-widget* 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 lacking or should be 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 *make-resizable* mixin from being applied to a *dialog%* subclass that already contains the *resize* method, and we wish to express this constraint in class contracts [Bracha 1992].

In response, we add an **absent** clause to our system of class contracts that lists fields and methods that should *not* be present in the contracted class. For example, here is a revised contract for *make-resizable*:

```
(define make-resizable/c
  (→ (class/c (absent resize)
              (inherit-field [width (≥/c 0)] [height (≥/c 0)])
              (inherit [undersize? (→* () ((≥/c 0) (≥/c 0)) boolean?)]))
     (class/c [resize (→dm [w (≥/c 0)] [h (≥/c 0)]
                           #:pre (not (send this undersize? w h))
                           void?)]
              (override [resize (→m (≥/c 0) (≥/c 0) void?)]))))
```

The **absent** clause specifies that the argument must not define a public method *resize*. If it does, the contract system signals a violation as soon as the mixin is applied.

Because Racket's class system supports augmentable methods, our contract system describes 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.

3.2. Contracts, Blame, and Dynamic Dispatch

In our system, 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 the call takes place. Furthermore, a method implementation may originate from a class defined in a separate module across a contract boundary. 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-resizable*, which is the same as *make-resizable* but checked with the *make-resizable/c* contract:

```
(define/contract checked-resizable make-resizable/c make-resizable)
(define resizable% (checked-resizable dialog%))
```

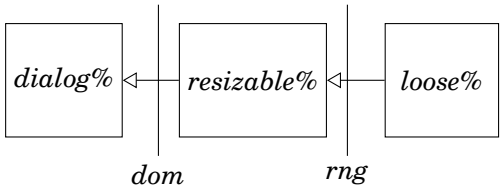
We also create a subclass of *resizable%* that overrides the definition of *resize*:

```
(define loose%
  (class resizable%
    (super-new)
    (inherit undersize?)
    (inherit-field width height min-width min-height)
    (define/override (resize w h)
      (cond [(not (undersize? w h)) (super resize)]
            [else (set! width min-width)
                  (set! height min-height)]))))

(define d4 (new loose% [label "D4"] [width 500]))
```

A call to *resize* on *loose%* resizes the dialog to its minimum size when an invalid size is given, taking advantage of the relaxed contract as described in the **override** clause.

Let *dom* be the domain contract of *make-resizable*/*c* and *rng* be the range contract. Assuming that the three class definitions exist in distinct contract regions, the contract boundaries between *dialog%*, *resizable%*, and *loose%* are as follows:



That is, *dom* mediates interactions between *dialog%* and its subclasses, and *rng* mediates interactions between *loose%* and its superclasses. Since *d4* instantiates *loose%*, method calls on it may trigger the contract checks in the class hierarchy of *loose%*.

If we call *resize* on *d4*:

(**send** *d4 resize* 200 200)

then the call to *undersize?* inside *loose%* uses the implementation from the superclass *dialog%*. The call is therefore checked according to the **inherit** clause in *dom*, because the call crosses the contract boundary between *dialog%* and *resizable%*. If we instead call *halve* on *d4*:

(**send** *d4 halve*)

then the *resize* method in *loose%* is called from *resizable%*. This call is checked according to the **override** clause in *rng*, since the call crosses the contract boundary between *resizable%* and *loose%*. In short, we get two different contract stories, depending on the path a method call takes through the dynamically constructed class hierarchy.

Since we do not know which implementation is invoked from a call due to dynamic dispatch, we cannot know which contracts are enforced until runtime. In turn, 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 mathematical model and its careful analysis.

4. THE MODEL

The class contract system introduces a complex strategy for retrieving and attaching contracts to method implementations upon method invocation. In particular, this strategy treats method calls differently depending on the relative position of the sender object in the class hierarchy and the invoked method. Furthermore, the method retrieval process affects the assignment of blame for contract violations. To clarify these points, we present a formal semantics for our contract system.

CPCF [Dimoulas and Felleisen 2011] is the starting point for our formalization. It extends Plotkin's call-by-value PCF [Plotkin 1977] with contracts for higher-order functions. We gradually add classes and other constructs to approximate the essence of Racket's contract system for first-class classes.

4.1. Adding First-Class Classes to CPCF

Figure 4 presents the syntax of CPCF. The language comes with three kinds of contracts κ: **any**, which never fails; **flat**(e), first-order checks on base values; or $\kappa_1 \ldots \mapsto$

κ_r, higher-order contracts on functions. A programmer can protect a term e with a contract κ with the monitor construct $\mathbf{mon}_j^{k,l}(\kappa, e)$. Such a monitor divides the program into two components: the term e, dubbed the server, and the context of $\mathbf{mon}_j^{k,l}(\kappa, e)$, dubbed the client. The monitor regulates the interaction of the two components, which are also called the parties of the contract. The labels k and l on the monitor are identifiers for the server and client, respectively. As Dimoulas et al. [2011] explain, 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. Notice, that our version of CPCF is untyped. Thus our syntax admits mallformed programs that when run lead to runtime type errors. Our semantics represents such errors as stuck states.

$$
\begin{array}{lrcl}
\textbf{Contracts} & \kappa & ::= & \textbf{any} \mid \textbf{flat}(e) \mid \kappa \ldots \mapsto \kappa \\
\textbf{Terms} & e & ::= & v \mid x \mid e\ e \ldots \mid e + e \mid e - e \mid e \wedge e \mid e \vee e \\
 & & \mid & \textbf{zero?}(e) \mid \textbf{if}\ e\ e\ e \mid \textbf{mon}_l^{l,l}(\kappa, e) \mid \textbf{error}_l^l \\
\textbf{Values} & v & ::= & 0 \mid 1 \mid -1 \mid \ldots \mid \lambda(x \ldots).e \mid \textbf{tt} \mid \textbf{ff}
\end{array}
$$

Fig. 4. CPCF syntax

The result of adding first-class classes and their contracts to CPCF is CFCC; see figure 5. 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, yet it includes the features of the class system that pose the greatest challenges to contract checking and blame assignment: 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 ι. 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 **object%**. Because of the omission of fields from the model, an object of a class is just a reference to a class value.

A method m in object e_o can be called in one of three ways: either directly via $\mathbf{send}(e_o, m, e_1 \ldots)$; internally from the class where the call occurs via $\mathbf{isend}^\iota(e_o, m, e_1 \ldots)$, which corresponds to the local use of class functions as described in section 3; or via a super call, $\mathbf{super}^\iota(e_o, m, e_1 \ldots)$. The latter two forms are annotated with the unique identifier ι 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. We keep the unique identifier of a class value inaccessible to the programmer, but equip all

[3]One may argue that our contract system could be derived through an encoding in an even smaller object calculus, but doing so would obscure the relation between blame and component boundaries in the model.

super and local method calls in the source code the default class identifier ‡ to keep the syntax uniform. As we show further on, the reduction semantics of CFCC employs a class identifier substitution meta-function to update class identifiers on super and local method calls once the enclosing class value is created.

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

$$\begin{array}{l}\textbf{class/c}\{\ \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1}\ldots \mapsto \kappa_r^{p_1})]\ldots \\ \qquad\quad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1}\ldots \mapsto \kappa_r^{o_1})]\ldots\}\end{array}$$

Public method contracts aggregate the public and inherited method contracts described in section 3. 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 them in guards. A guard 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.[4]

$$\begin{array}{lll}
\textbf{Contracts} & \kappa & ::= \ldots \mid \textbf{class/c}\{\ \textbf{public}\ [m\ (\kappa\ \kappa\ldots \mapsto \kappa)]\ldots \\
 & & \qquad\qquad \textbf{override}\ [m\ (\kappa\ \kappa\ldots \mapsto \kappa)]\ldots\} \\
\textbf{Terms} & e & ::= \ldots \mid c \mid \textbf{new}(e) \mid \textbf{send}(e, m, e\ldots) \\
 & & \quad \mid \textbf{isend}^{\iota}(e, m, e\ldots) \mid \textbf{super}^{\iota}(e, m, e\ldots) \\
\textbf{Values} & v & ::= \ldots \mid \gamma \\
\textbf{Class Values} & cv & ::= \textbf{object\%} \mid \textbf{class/v}^{\iota}\{\ v\ \textbf{methods}\ [m(this\ x\ldots)\ e]\ldots\} \\
\textbf{Classes} & c & ::= \textbf{class}\{\ e \\
 & & \qquad \textbf{inherit}\ m\ldots \\
 & & \qquad \textbf{public}\ [m(this\ x\ldots)\ e]\ldots \\
 & & \qquad \textbf{override}\ [m(this\ x\ldots)\ e]\ldots\} \\
\textbf{Objects} & o & ::= \textbf{object}(\gamma) \\
\textbf{Guards} & \gamma & ::= o \mid cv \mid \textbf{G}\{\ v \\
 & & \qquad\qquad \textbf{public}\ [m\ (\kappa\ \kappa\ldots \mapsto \kappa)_l^{l,l}]\ldots \\
 & & \qquad\qquad \textbf{override}\ [m\ (\kappa\ \kappa\ldots \mapsto \kappa)_l^{l,l}]\ldots\}
\end{array}$$

Fig. 5. CFCC syntax

4.2. CFCC Semantics

We next equip CFCC with a reduction semantics [Felleisen et al. 2009]. Each reduction rule deterministically decomposes a program into an evaluation context and a redex. The single-step reduction relation, $\hookrightarrow$, 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-reflexive 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; see figure 6.

Figure 7 presents the reduction rules for CFCC. The reduction relation consists of three groups of rules: the PCF standard declarative reductions, contract related reductions and object oriented reductions. There is also a program-global rule for handling programs that produce $\textbf{error}_j^k$ and return the error as the answer of the program.

[4]Guards correspond to chaperones and impersonators in the Racket implementation [Strickland et al. 2012].

$$
\begin{array}{rcl}
E & ::= & E\ e \mid v\ E \mid E + e \mid v + E \mid E - e \mid v - E \mid E \wedge e \mid v \wedge E \mid E \vee e \mid v \vee E \\
& \mid & \mathbf{zero?}(E) \mid \mathbf{if}\ E\ e\ e \mid \mathbf{mon}_j^{l,k}(\kappa, E) \mid \mathbf{new}(E) \mid \mathbf{send}(E, m, e \ldots) \\
& \mid & \mathbf{send}(v, m, v \ldots E e \ldots) \mid \mathbf{isend}^{\iota}(E, m, e \ldots) \mid \mathbf{isend}^{\iota}(v, m, v \ldots E e \ldots) \\
& \mid & \mathbf{super}^{\iota}(E, m, e \ldots) \mid \mathbf{super}^{\iota}(v, m, v \ldots E e \ldots) \mid \mathbf{class}\{\ E \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{inherit}\ m \ldots \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{public}\ [m(this\ x \ldots)\ e] \ldots \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{override}\ [m(this\ x \ldots)\ e] \ldots \}
\end{array}
$$

Fig. 6. Evaluation contexts

Contract checking in CFCC proceeds along the lines of Findler and Felleisen [2002]. Monitors of flat contracts, $\mathbf{flat}(e)$, are expanded to `if`-expressions 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 with a monitor of the corresponding precondition contract κ_i and then applies the monitored arguments to the original function. The result of the application is guarded with the post-condition contract κ_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 corresponding guard that wraps the value and contains the method contract. Moreover, each guard 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. For inherited and public method contracts, the server is the superclass that provides the method, and the clients are the subclasses that use the method. For overridden methods the situation is reversed. An override contract is a request from the superclass that uses a method to the subclasses that implement it.

The object-oriented reductions are the core of our model and we dedicate the following sub-section to examine each reduction rule separately.

4.3. Object-Oriented Reductions for CFCC

The reductions for the object-oriented constructs of the language depend on a number of meta-functions (figure 8). In the following paragraphs we briefly discuss them as we describe the rules, but here we provide only the definitions of the major meta-functions to keep the presentation concise. The interested reader can find the definitions of all the remaining meta-functions in appendix A.

A new object of a class can be created using the $\mathbf{new}(e)$ construct if the argument evaluates to a class value. The $\mathcal{C}V$ meta-function (appendix A.1) 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 $\mathcal{M}ethods$ meta-function (appendix A.2) collects all the method names in a class hierarchy starting from the given class value or object. The reduction rule produces a fresh class identi-

$E[\cdots]$	$\hookrightarrow E[\cdots]$	
$n_1 + n_2$	$\cdot\ n$	where $n_1 + n_2 = n$
$n_1 - n_2$	$\cdot\ n$	where $n_1 - n_2 = n$
zero?(0)	$\cdot$ **tt**	
zero?(n)	$\cdot$ **ff**	if $n \neq 0$
$v_1 \wedge v_2$	$\cdot\ v$	where $v_1 \wedge v_2 = v$
$v_1 \vee v_2$	$\cdot\ v$	where $v_1 \vee v_2 = v$
if tt $e_1\ e_2$	$\cdot\ e_1$	
if ff $e_1\ e_2$	$\cdot\ e_2$	
$\lambda(x_1 \ldots x_n).e\ v_1 \ldots v_n$	$\cdot\ \{v_1 \ldots v_n / x_1 \ldots x_n\}e$	
$\mathbf{mon}_j^{k,l}(\kappa_1 \ldots \kappa_n \mapsto \kappa_r, v)$	$\cdot\ \lambda(x_1 \ldots x_n).$ $\mathbf{mon}_j^{k,l}(\kappa_r,$ $v\ \mathbf{mon}_j^{l,k}(\kappa_1, x_1) \ldots \mathbf{mon}_j^{l,k}(\kappa_n, x_n))$	
$\mathbf{mon}_j^{k,l}(\mathbf{any}, v)$	$.\ v$	
$\mathbf{mon}_j^{k,l}(\mathbf{flat}(e), v)$	. **if** $(e\ v)\ v$ $\mathbf{error}_j^k$	
$\mathbf{mon}_j^{k,l}(\kappa, \gamma)$	. $\mathbf{G}\{\ \gamma$ $\mathbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots$ $\mathbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}$	
where $\kappa =$ **class/c**$\{$ **public** $[m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})] \ldots$ **override** $[m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})] \ldots\}$		
new(γ)	. **object**(γ)	if $\mathcal{CV}[\![\gamma]\!]$
class$\{\ \gamma$ **inherit** $m_{i_1} \ldots$ **public** $[m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ e_{p_1}] \ldots$ **override** $[m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ e_{o_1}] \ldots\}$	. **class/v**$^\iota\{\ \gamma$ **methods** $[m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ \{\iota/^{id}_{\ddagger}\}e_{p_1}] \ldots$ $[m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ \{\iota/^{id}_{\ddagger}\}e_{o_1}] \ldots\ \}$	
where ι fresh (does not occur in the eval. context filled with the entire redex) if $\mathcal{CV}[\![\gamma]\!]$, $\forall_j\ m_{i_j}, m_{o_j} \in \mathcal{M}ethods[\![\gamma]\!]$, $\forall_j\ m_{p_j} \notin \mathcal{M}ethods[\![\gamma]\!]$		
send$(\gamma, m, v \ldots)$ where $e = \mathcal{P}ull[\![\gamma, m]\!]$ and if $\mathcal{O}bject[\![\gamma]\!]$, $m \in \mathcal{M}ethods[\![\gamma]\!]$	$.\ e\ \gamma\ v \ldots$	
super$^\iota(\gamma, m, v \ldots)$ where $e = \mathcal{P}ull[\![cv, m]\!]$, $cv = \mathcal{G}etS[\![\gamma, \iota]\!]$ and if $\mathcal{O}bject[\![\gamma]\!]$, $\mathcal{I}s[\![\gamma, \iota]\!]$, $m \in \mathcal{M}ethods[\![cv, \iota]\!]$	$.\ e\ \gamma\ v \ldots$	
isend$^\iota(\gamma, m, v \ldots)$ where $e = \mathcal{F}ind[\![\gamma, \iota, m]\!]$ and if $\mathcal{O}bject[\![\gamma]\!]$, $\mathcal{I}s[\![\gamma, \iota]\!]$, $m \in \mathcal{M}ethods[\![\mathcal{G}et[\![\gamma, \iota]\!]]\!]$	$.\ e\ \gamma\ v \ldots$	
$E^l[\mathbf{error}_j^k]$	$\longmapsto \mathbf{error}_j^k$	

Fig. 7. Reduction semantics for CFCC

fier for the new class value. The special substitution function $\{\iota/^{id}_{\ddagger}\}e$ (appendix A.3) replaces all occurrences of $\ddagger$ annotations on super and local method calls in e with ι.

The **send**$(e_o, m, e_1 \ldots)$ expression performs a method send to an object if e_o evaluates to an object v and m is a method defined in the class hierarchy that v specifies. 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 uses the meta-functions $\mathcal{O}bject$ (appendix A.4) and $\mathcal{M}ethods$ to determine if the method call is valid. Then it employs the meta-function $\mathcal{P}ull$ from figure 9. This meta-function traverses the class value of

Major Meta-functions	
$\mathcal{P}ull$	$\gamma \times m \to e$ traverses γ, retrieves the first implementation of m and brings the implementation back to the starting point of the traversal while attaching all the contracts for m along the way
$\mathcal{P}ush$	$\gamma \times e \times \iota \times m \to e$ traverses γ until it retrieves class value ι and attaches to e any contracts for m along the way
$\mathcal{F}ind$	$\gamma \times \iota \times m \to e$ performs method retrieval for local method calls of m; traverses γ until it either finds an implementation for m or class value ι, then it delegates method retrieval to $\mathcal{P}ush$ or $\mathcal{P}ull$ respectively
Helper Meta-functions	
$\mathcal{C}V$	$v \to \mathcal{T}rue$ or $\mathcal{F}alse$ checks if v is a class value
$\mathcal{M}ethods$	$\gamma \to \{m\}$ returns the set of methods of γ
$\{-/^{id}-\}-$	$\iota \times \iota \times e \to e$ substitutes the first id for the second in e respecting class scoping
$\mathcal{O}bject$	$v \to \mathcal{T}rue$ or $\mathcal{F}alse$ checks if v is an object
$\mathcal{I}s$	$\gamma \times \iota \to \mathcal{T}rue$ or $\mathcal{F}alse$ checks if γ implements or extends class ι
$\mathcal{G}etS$	$\gamma \times \iota \to cv$ returns the super class of class ι after traversing γ
$\mathcal{G}et$	$\gamma \times \iota \to cv$ returns class ι after traversing γ

Fig. 8. Meta-functions summary

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 $\textbf{super}^{\iota}(e_o, m, e_1 \ldots)$ 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 $\mathcal{G}etS$ meta-function (appendix A.6) to obtain cv. This meta-function traverses o until it runs into a class value with class identifier ι 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 $\mathcal{I}s$ (appendix A.5) that checks if a given object implements a class with identifier ι. After obtaining cv, the rule applies $\mathcal{P}ull$ on cv to get the contracted implementation of m.

Local method calls have the shape $\textbf{isend}^{\iota}(e_o, m, e_1 \ldots)$ 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 $\mathcal{G}et$ retrieves cv (figure A.7), and the meta-function $\mathcal{F}ind$ (figure 10) constructs the result of the reduction. The latter ascends the class hierarchy starting from the given object until it finds either cv or the first implementation of m. If it finds cv, it returns the result of using $\mathcal{P}ull$ on cv and m. Otherwise, it delegates the task to $\mathcal{P}ush$ (figure 11). This last meta-function pushes the implementation of m up the class hierarchy until it reaches cv and then returns the result. The initially detected implementation of m 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.

$$
\begin{array}{lcl}
\mathcal{P}ull[\![\gamma, m]\!] & = & \mathcal{P}ull[\![\gamma', m]\!] \\
 & & \textbf{if}\, m \notin \{m_1, \ldots\} \\
 & & \textbf{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(\mathit{this}_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \\
\mathcal{P}ull[\![\gamma, m]\!] & = & \lambda(\mathit{this}\ x_1 \ldots).e \\
 & & \textbf{if}\, [m(\mathit{this}\ x_1 \ldots)\ e] \in \{[m_1(\mathit{this}_1\ x_1^1 \ldots)\ e_1], \ldots\} \\
 & & \textbf{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(\mathit{this}_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \\
\mathcal{P}ull[\![\textbf{object}(\gamma), m]\!] & = & \mathcal{P}ull[\![\gamma, m]\!] \\
 & & \\
\mathcal{P}ull[\![\gamma, m]\!] & = & \mathcal{P}ull[\![\gamma', m]\!] \\
 & & \textbf{if}\, m \notin \{m_{p_1}, \ldots\} \\
 & & \textbf{where}\, \gamma = \mathbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public}\ [m_{p_1}\ (\kappa_{\mathit{this}}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override}\ [m_{o_1}\ (\kappa_{\mathit{this}}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
 & & \\
\mathcal{P}ull[\![\gamma, m]\!] & = & \textbf{mon}_j^{k,l}(\kappa_{\mathit{this}}\ \kappa_1 \ldots \mapsto \kappa_r, \mathcal{P}ull[\![\gamma', m]\!]) \\
 & & \textbf{if}\, [m\ (\kappa_{\mathit{this}}\ \kappa_1 \ldots \mapsto \kappa_r)_j^{k,l}] \in \{[m_{p_1}\ (\kappa_{\mathit{this}}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}], \ldots\} \\
 & & \textbf{where}\, \gamma = \mathbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public}\ [m_{p_1}\ (\kappa_{\mathit{this}}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override}\ [m_{o_1}\ (\kappa_{\mathit{this}}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

Fig. 9. The $\mathcal{P}ull$ meta-function

Before concluding this section, we demonstrate how our model works through the *checked-resizable* example from section 3. To reduce the clutter in the example, which involves a series of class definitions, we represent only the important elements:

$$
\begin{array}{lcl}
\mathit{d4} & = & \textbf{new}(\mathit{loose}\%) \\
\mathit{loose}\% & = & \textbf{class}\{\ \mathit{resizable}\% \\
 & & \quad \textbf{inherit}\ \mathit{undersize}? \\
 & & \quad \textbf{override}\ \mathit{resize}\ (\mathit{this}\ w\ h)\ \ldots \textbf{isend}^{\ddagger}(\mathit{this}, \mathit{undersize}?, w, h) \ldots\} \\
\mathit{resizable}\% & = & \mathit{checked\text{-}resizable}\ \ \mathit{dialog}\% \\
\mathit{checked\text{-}resizable} & = & \textbf{mon}_j^{k,l_o}(\mathit{make\text{-}resizable}/c, \mathit{make\text{-}resizable}) \\
\mathit{make\text{-}resizable}/c & = & \mathit{dom} \mapsto \mathit{rng} \\
\mathit{make\text{-}resizable} & = & \lambda(\mathit{base}).\textbf{class}\{\ \mathit{base} \\
 & & \qquad\qquad \textbf{inherit}\ \mathit{undersize}? \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\quad \mathit{resize}\ (\mathit{this}\ w\ h) \ldots \mathit{halve}\ (\mathit{this}) \ldots \\
 & & \qquad\qquad\quad \textbf{isend}^{\ddagger}(\mathit{this}, \mathit{resize}, w, h) \ldots \qquad\qquad \} \\
\mathit{dom} & = & \textbf{class/c}\{\ \textbf{public}\ \mathit{undersize}?\ (\ldots \mapsto \ldots)\} \\
\mathit{rng} & = & \textbf{class/c}\{\ \textbf{public}\ \mathit{resize}\ (\ldots \mapsto \ldots) \\
 & & \qquad\qquad \textbf{override}\ \mathit{resize}\ (\ldots \mapsto \ldots)\}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{F}ind[\![\gamma, \iota, m]\!] & = & \mathcal{P}ull[\![\gamma, m]\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \\
\mathcal{F}ind[\![\gamma, \iota, m]\!] & = & \mathcal{P}ush[\![\gamma', \lambda(this_j\ x_1^j \ldots).e_j, \iota, m]\!] \\
 & & \text{if } [m(this\ x_1 \ldots)\ e] \in \{[m_1(this_1\ x_1^1 \ldots)\ e_1], \ldots\} \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \text{and } \iota \neq \iota' \\
 & & \\
\mathcal{F}ind[\![\gamma, \iota, m]\!] & = & \mathcal{F}ind[\![\gamma', \iota, m]\!] \\
 & & \text{if } m \notin \{m_1, \ldots\} \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \text{and } \iota \neq \iota' \\
 & & \\
\mathcal{F}ind[\![\textbf{object}(\gamma), \iota, m]\!] & = & \mathcal{F}ind[\![\gamma, \iota, m]\!] \\
 & & \\
\mathcal{F}ind[\![\gamma, \iota, m]\!] & = & \mathcal{F}ind[\![\gamma', \iota, m]\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

Fig. 10. The $\mathcal{F}ind$ meta-function

We first create an object $d4$ of class $loose\%$. Class $loose\%$ is a subclass of class $resizable\%$. In turn the latter class is the result of applying the contract protected mixin $checked\text{-}resizable$ to $dialog\%$. As in section 3, the pieces of the mixin contract regulate the flow of values in the class hierarchy behind $d4$. More specifically, dom monitors interactions between $dialog\%$ and its subclasses, while rng monitors interactions between $loose\%$ and its superclasses:

$$
\begin{array}{l}
\textbf{mon}_j^{k,l_o}(rng, \textbf{class/v}^{\iota}\{\ \textbf{mon}_j^{l_o,k}(dom, dialog\%) \\
\qquad\qquad \textbf{methods} \\
\qquad\qquad\quad resize\ (this\ w\ h)\ \ldots \\
\qquad\qquad\quad halve\ (this)\ \ldots \textbf{isend}^{\iota}(this, resize, w, h) \ldots\ \})
\end{array}
$$

In this setting, the term $\textbf{send}(d4, resize, 200,\ 200)$ delegates the retrieval of the implementation of $resize$ to $\mathcal{P}ull$ which returns the body of $resize$ from $loose\%$. No contract boundary crossing takes place, thus the meta-function wraps no contract around the implementation. The evaluation of the body of the method leads to the evaluation of the local method call $\textbf{isend}^{\iota}(d4, undersize?, w, h)$ where ι is the class identifier for class $loose\%$. The meta-function $\mathcal{F}ind$ is responsible for delivering the correct implementation of $undersize?$. It does so by traversing the class hierarchy starting from $loose\%$ and looking either for an implementation of $undersize?$ or for a class value with class id ι. It runs into the latter first, so this is the case of an inherited method. Hence, it delegates

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\gamma, e, \iota, m]\!] & = & e \\
& & \text{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
& & \qquad \textbf{methods} \\
& & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\gamma, e, \iota, m]\!] & = & \mathcal{P}ush[\![\gamma', e, \iota, m]\!] \\
& & \text{where}\, \gamma = \textbf{class/v}^{\iota'}\{\ \gamma' \\
& & \qquad \textbf{methods} \\
& & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
& & \text{and}\, \iota \neq \iota'
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\gamma, e, \iota, m]\!] & = & \mathcal{P}ush[\![\gamma', e, \iota, m]\!] \\
& & \text{if}\, m \notin \{m_{o_1}, \ldots\} \\
& & \text{where}\, \gamma = \textbf{G}\{\ \gamma' \\
& & \qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
& & \qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\gamma, e, \iota, m]\!] & = & \mathcal{P}ush[\![\gamma', \textbf{mon}_j^{k,l}(\kappa_{this}\ \kappa_1 \ldots \mapsto \kappa_r, e), \iota, m]\!] \\
& & \text{if}\, [m\ (\kappa_{this}\ \kappa_1 \ldots \mapsto \kappa_r)_j^{k,l}] \in \{[m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{k,l}], \ldots\} \\
& & \text{where}\, \gamma = \textbf{G}\{\ \gamma' \\
& & \qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{l,k}] \ldots \\
& & \qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{k,l}] \ldots\}
\end{array}
$$

Fig. 11. The $\mathcal{P}ush$ meta-function

the method implementation retrieval to $\mathcal{P}ull$. This meta-function starts from the class $resizable\%$ and ascends the class hierarchy up to $dialog\%$ where it locates the closest implementation of the method. It then pulls the implementation back to the call site. The meta-function also wraps the implementation with all the inherit contracts for the method it encounters along the way. In this case, there is only one such contract; the one in dom. It is worth mentioning finally that the monitor for this contract has l_o as the server label and k as the client one due to the label shuffling of the mixin contract decomposition.

The second term of the example, $\textbf{send}(d4, halve)$, behaves in a different way. Again $\mathcal{P}ull$ locates directly the implementation of $halve$. The evaluation proceeds with the local method call $\textbf{isend}^{\iota_1}(d4, resize, w, h)$, where ι_1 is the class id for $resizeable\%$. The call uses $\mathcal{F}ind$ to discover the right implementation of $resize$. However, $\mathcal{F}ind$ locates the implementation of the method in $loose\%$ before finding a class with class id ι_1. Thus, this is an overridden method scenario and $\mathcal{F}ind$ passes the task of attaching contracts to the implementation to $\mathcal{P}ush$. The meta-function pushes the implementation until the class value of $resizable\%$ and wraps it with the appropriate contract, the override contract from rng. Again, the monitor that surfaces at the call has l_o as the server label and k as the client one. This time, however, the label assignment is due to the way the class contract reduction rule picks the labels for contracts on overridden methods.

Note: As mentioned at the beginning of this section, our model has only a subset of the features of the Racket class and contract systems. However, the missing features do not undermine the value of the model. Our model adequately expresses all the issues with contract checking and blame assignment for first-class classes. In particular, the main discrepancies between the Racket implementation and the model are the lack of

fields, Beta-style inheritance, and inheritance contracts in the model. In our model we merge inheritance and public contracts because the contract system treats them in a similar manner. The only difference is a set of trivial first-order checks upon contract application that guarantee that inherit contracts are applied to methods that a class inherits and does not re-implement.

5. CORRECT BLAME: THE BUILD-UP

The most fundamental question about a higher-order contract system is whether it assigns blame correctly. To give a satisfying answer, we must formally define and prove blame correctness [Dimoulas et al. 2011]. The idea is that a contract specifies constraints on the interaction between a server and its clients. Each party 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 partners, the values are checked against the obligations of the party. If the values do not meet the specifications, the party is blamed for violating the contract.

INFORMAL DEF. 5.1 (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.*

CFCC is the starting point for our formalization of of blame correctness but it does not carry enough information. We therefore decorate the syntax of our model with two kinds of annotations: obligations and ownership. These extra bits of information are sufficient to state the definition of blame correctness and to prove our design blame correct.

5.1. CFCC with Obligations and Ownership Annotations

The obligations of each party are a subset of its flat pieces of the contract, i.e. the predicates that the party's flat values must satisfy. 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 does not impose any obligations.

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 contract monitors blame only parties that send values into the corresponding contract. CFCC uses ownership annotations $\|e\|^l$ to designate that a party l owns a term e, and obligation annotations $\lfloor \mathbf{flat}(e_c) \rfloor^l$ to mark that a party l is responsible for a flat contract $\mathbf{flat}(e_c)$.

$$
\begin{array}{lrcl}
\textbf{Contracts} & \kappa & ::= & \ldots \mid \lfloor \mathbf{flat}(e) \rfloor^l \\
\textbf{Terms} & e & ::= & \ldots \mid \|e\|^l \\
\textbf{Values} & v & ::= & \ldots \mid \|v\|^l
\end{array}
$$

Fig. 12. CPCF syntax with Ownership and Obligations

Not all annotations are meaningful in the source code, however. Specifically, ownership annotations must coincide with component boundaries. Contract monitors also indicate component boundaries. Hence, in the source code, ownership annotations 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 13 lists the complete set of rules. The most interesting rule concerns monitor expressions:

$$\frac{k \vdash e \qquad k; l; j \rhd \kappa}{l \vdash \mathbf{mon}_j^{k,l}(\kappa, \|e\|^k)}$$

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 $k; l; j \rhd \kappa$.

The rules for well-formed contracts in figure 14 check whether the client and server labels on the monitor sit on the appropriate negative and positive flat parts of a contract. Furthermore, the rules demand that the owner of the predicates in all the flat contracts of κ is the contract j. The rule for the **any** contract has no premises as the **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 override 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 local method calls and super calls to be the default class identifier ‡.

$\boxed{l \vdash e}$

$$\frac{n \in \mathit{Integers}}{l \vdash n} \qquad \frac{}{l \vdash \mathbf{tt}} \qquad \frac{}{l \vdash \mathbf{ff}} \qquad \frac{l \vdash e_1}{l \vdash \mathbf{zero?}(e_1)}$$

$$\frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 + e_2} \qquad \frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 - e_2} \qquad \frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 \wedge e_2} \qquad \frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 \vee e_2}$$

$$\frac{}{l \vdash x} \qquad \frac{l \vdash e}{l \vdash \lambda(x_1 \ldots).e} \qquad \frac{l \vdash e_f \quad l \vdash e_1 \quad \ldots}{l \vdash e_f\ e_1 \ldots} \qquad \frac{k \vdash e \quad k; l; j \rhd \kappa}{l \vdash \mathbf{mon}_j^{k,l}(\kappa, \|e\|^k)}$$

$$\frac{l \vdash e_1 \quad l \vdash e_2 \quad l \vdash e_3}{l \vdash \mathbf{if}\ e_1\ e_2\ e_3} \qquad \frac{}{l \vdash \mathbf{object\%}} \qquad \frac{l \vdash e_o \quad l \vdash e_1 \quad \ldots}{l \vdash \mathbf{super}^{\ddagger}(e_o, m, e_1 \ldots)}$$

$$\frac{l \vdash e}{l \vdash \mathbf{new}(e)} \qquad \frac{l \vdash e_o \quad l \vdash e_1 \quad \ldots}{l \vdash \mathbf{send}(e_o, m, e_1 \ldots)} \qquad \frac{l \vdash e_o \quad l \vdash e_1 \quad \ldots}{l \vdash \mathbf{isend}^{\ddagger}(e_o, m, e_1 \ldots)}$$

$$\frac{l \vdash e_s \quad l \vdash e_{p_1} \quad \ldots \quad l \vdash e_{o_1} \quad \ldots}{\begin{array}{l} l \vdash \mathbf{class}\{\ e_s \\ \quad \mathbf{inherit}\ m_{i_1} \ldots \\ \quad \mathbf{public}\ [m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ e_{p_1}] \ldots \\ \quad \mathbf{override}\ [m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ e_{o_1}] \ldots \} \end{array}}$$

Fig. 13. Ownership coincides with contract monitors

5.2. Semantics for CFCC with Annotations

In CFCC with annotations, the reduction relation, $\longmapsto$, is the means not only for evaluating terms but also for propagating ownership annotations as values flow from one component to another.

The first step to achieve ownership propagation is to modify the evaluation contexts of the language. The grammar of evaluation contexts in figure 15 is similar to that for

$\boxed{k; l; j \rhd \kappa}$

$$\frac{l; k; j \rhd \kappa_1 \quad \ldots \quad l; k; j \rhd \kappa_n \quad k; l; j \rhd \kappa_r}{k; l; j \rhd \kappa_1 \ldots \kappa_n \mapsto \kappa_r}$$

$$\frac{}{k; l; j \rhd \textbf{any}} \qquad \frac{j \vdash e}{k; l; j \rhd \lfloor \textbf{flat}(\|e\|^j) \rfloor^k}$$

$$\frac{\begin{array}{l} k; l; j \rhd \kappa_{this}^{p_1} \ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1} \quad \ldots \\ l; k; j \rhd \kappa_{this}^{o_1} \ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1} \quad \ldots \end{array}}{\begin{array}{l} k; l; j \rhd \ \textbf{class/c}\{ \ \textbf{public} \ [m_{p_1} \ (\kappa_{this}^{p_1} \ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})] \ldots \\ \qquad\qquad \textbf{override} \ [m_{o_1} \ (\kappa_{this}^{o_1} \ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{p_1})] \ldots \} \end{array}}$$

Fig. 14. Obligations coincide with labels on monitors

$$\begin{array}{rcl}
E^l & ::= & E^l \ e \mid v \ E^l \mid E^l + e \mid v + E^l \\
& \mid & E^l - e \mid v - E^l \mid E^l \wedge e \mid v \wedge E^l \\
& \mid & E^l \vee e \mid v \vee E^l \mid \textbf{zero?}(E^l) \\
& \mid & \textbf{if} \ E^l \ e \ e \mid \textbf{mon}_j^{l,k}(\kappa, E^{l_o}) \\
& \mid & \textbf{mon}_j^{l',k}(\kappa, E^l), \ l \neq l_o \\
& \mid & \|E^{l_o}\|^l \mid \|E^l\|^{l'}, \ l \neq l_o \\
& \mid & \textbf{new}(E^l) \mid \textbf{send}(E^l, m, e \ldots) \\
& \mid & \textbf{send}(v, m, v \ldots E^l e \ldots) \\
& \mid & \textbf{isend}^\iota(E^l, m, e \ldots) \\
& \mid & \textbf{isend}^\iota(v, m, v \ldots E^l e \ldots) \\
& \mid & \textbf{super}^\iota(E^l, m, e \ldots) \\
& \mid & \textbf{super}^\iota(v, m, v \ldots E^l e \ldots) \\
& \mid & \textbf{class}\{ \ E^l \\
& & \quad \textbf{inherit} \ m \ldots \\
& & \quad \textbf{public} \ [m(this \ x \ldots) \ e] \ldots \\
& & \quad \textbf{override} \ [m(this \ x \ldots) \ e] \ldots \}
\end{array}$$

$$\begin{array}{rcl}
E^{l_o} & ::= & [\ ] \mid E^{l_o} \ e \mid v \ E^{l_o} \mid E^{l_o} + e \mid v + E^{l_o} \\
& \mid & E^{l_o} - e \mid v - E^{l_o} \mid E^{l_o} \wedge e \mid v \wedge E^{l_o} \\
& \mid & E^{l_o} \vee e \mid v \vee E^{l_o} \mid \textbf{zero?}(E^{l_o}) \\
& \mid & \textbf{if} \ E^{l_o} \ e \ e \\
& \mid & \textbf{new}(E^{l_o}) \mid \textbf{send}(E^{l_o}, m, e \ldots) \\
& \mid & \textbf{send}(e, m, v \ldots E^{l_o} e \ldots) \\
& \mid & \textbf{isend}^\iota(E^{l_o}, m, e \ldots) \\
& \mid & \textbf{isend}^\iota(v, m, v \ldots E^{l_o} e \ldots) \\
& \mid & \textbf{super}^\iota(E^{l_o}, m, e \ldots) \\
& \mid & \textbf{super}^\iota(v, m, v \ldots E^{l_o} e \ldots) \\
& \mid & \textbf{class}\{ \ E^{l_o} \\
& & \quad \textbf{inherit} \ m \ldots \\
& & \quad \textbf{public} \ [m(this \ x \ldots) \ e] \ldots \\
& & \quad \textbf{override} \ [m(this \ x \ldots) \ e] \ldots \}
\end{array}$$

Fig. 15. Parameterized evaluation contexts

CFCC but it 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 prove in section 6 that the two always coincide. We use E^{l_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, l_o. The label parameter plays a key role in keeping track of ownership during reduction.

Figure 16 presents the reduction rules for CFCC with annotations. Each rule corresponds to a rule for CFCC without annotations (figure 7). The corresponding rules are morphologically identical, modulo ownership annotations, and have the same computational content. However, the rules for CFCC with annotations come with an additional level of ownership label manipulation and propagation. The latter do not affect computation per se; from the perspective of program evaluation they play a purely decorative role as we prove at the end of this section. Nevertheless, the ownership propagation within the semantics allows us to reason about the flow of values between

$E^l[\cdots]$	$\longmapsto E^l[\cdots]$	
$\|n_1\|^{\vec{k}} + \|n_2\|^{\vec{l}}$	$\cdot\ n$	where $n_1 + n_2 = n$
$\|n_1\|^{\vec{k}} - \|n_2\|^{\vec{l}}$	$\cdot\ n$	where $n_1 - n_2 = n$
$\mathbf{zero?}(\|0\|^{\vec{l}})$	$\cdot\ \mathbf{tt}$	
$\mathbf{zero?}(\|n\|^{\vec{k}})$	$\cdot\ \mathbf{ff}$	if $n \neq 0$
$\|v_1\|^{\vec{k}} \wedge \|v_2\|^{\vec{l}}$	$\cdot\ v$	where $v_1 \wedge v_2 = v$
$\|v_1\|^{\vec{k}} \vee \|v_2\|^{\vec{l}}$	$\cdot\ v$	where $v_1 \vee v_2 = v$
$\mathbf{if}\ \|\mathbf{tt}\|^{\vec{l}}\ e_1\ e_2$	$\cdot\ e_1$	
$\mathbf{if}\ \|\mathbf{ff}\|^{\vec{l}}\ e_1\ e_2$	$\cdot\ e_2$	
$\|\lambda(x_1 \ldots x_n).e\|^{\vec{l}}\ v_1 \ldots v_n$	$\cdot\ \|\{\|v_1\|^{l \overleftarrow{l}} \ldots \|v_n\|^{l \overleftarrow{l}} / x_1 \ldots x_n\} e\|^{\vec{l}}$	
$\mathbf{mon}_j^{k,l}(\kappa_1 \ldots \kappa_n \mapsto \kappa_r, v)$	$\cdot\ \lambda(x_1 \ldots x_n).$ $\mathbf{mon}_j^{k,l}(\kappa_r,$ $v\ \mathbf{mon}_j^{l,k}(\kappa_1, x_1) \ldots \mathbf{mon}_j^{l,k}(\kappa_n, x_n))$	
$\mathbf{mon}_j^{k,l}(\mathbf{any}, v)$	$.\ v$	
$\mathbf{mon}_j^{k,l}(\lfloor \mathbf{flat}(e) \rfloor^{l'}, v)$	$.\ \mathbf{if}\ (e\ v)\ v\ \mathbf{error}_j^k$	
$\mathbf{mon}_j^{k,l}(\kappa, \|\gamma\|^{\vec{l}})$	$.\ \mathbf{G}\{\ \|\gamma\|^{\vec{l}}$ $\mathbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots$ $\mathbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}$	
where $\kappa =$ $\mathbf{class/c}\{\ \mathbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})] \ldots$ $\mathbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})] \ldots\}$		
$\mathbf{new}(\|\gamma\|^{\vec{l}})$	$.\ \mathbf{object}(\|\gamma\|^{\vec{l}})$	if $\mathcal{CV}[\![\|\gamma\|^{\vec{l}}]\!]$
$\mathbf{class}\{\ \|\gamma\|^{\vec{l}}$ $\mathbf{inherit}\ m_{i_1} \ldots$ $\mathbf{public}\ [m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ e_{p_1}] \ldots$ $\mathbf{override}\ [m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ e_{o_1}] \ldots\}$	$.\ \mathbf{class/v}^{\iota}\{\ \|\gamma\|^{\vec{l}}$ $\mathbf{methods}$ $[m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ \{\iota/^{id}_{\ddagger}\} e_{p_1}] \ldots$ $[m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ \{\iota/^{id}_{\ddagger}\} e_{o_1}] \ldots\}$	
where ι fresh (does not occur in the eval. context filled with the entire redex)		
if $\mathcal{CV}[\![\|\gamma\|^{\vec{l}}]\!], \forall_j\ m_{i_j}, m_{o_j} \in \mathcal{M}ethods[\![\|\gamma\|^{\vec{l}}]\!], \forall_j\ m_{p_j} \notin \mathcal{M}ethods[\![\|\gamma\|^{\vec{l}}]\!]$		
$\mathbf{send}(\|\gamma\|^{\vec{l}}, m, v \ldots)$	$.\ e\ \|\gamma\|^{\vec{l}}\ v \ldots$	
where $e = \mathcal{P}ull[\![\|\gamma\|^{\vec{l}}, m]\!]$ and if $\mathcal{O}bject[\![\|\gamma\|^{\vec{l}}]\!], m \in \mathcal{M}ethods[\![\|\gamma\|^{\vec{l}}]\!]$		
$\mathbf{super}^{\iota}(\|\gamma\|^{\vec{l}}, m, v \ldots)$	$.\ e\ \|\gamma\|^{\vec{l}}\ v \ldots$	
where $e = \mathcal{P}ull[\![v', m]\!], v' = \mathcal{G}etS[\![\|\gamma\|^{\vec{l}}, \iota, l]\!]$ and if $\mathcal{O}bject[\![\|\gamma\|^{\vec{l}}]\!], \mathcal{I}s[\![\|\gamma\|^{\vec{l}}, \iota]\!], m \in \mathcal{M}ethods[\![v']\!]$		
$\mathbf{isend}^{\iota}(\|\gamma\|^{\vec{l}}, m, v \ldots)$	$.\ e\ \|\gamma\|^{\vec{l}}\ v \ldots$	
where $e = \mathcal{F}ind[\![\|\gamma\|^{\vec{l}}, \iota, m, l]\!]$ and if $\mathcal{O}bject[\![\|\gamma\|^{\vec{l}}]\!], \mathcal{I}s[\![\|\gamma\|^{\vec{l}}, \iota]\!], m \in \mathcal{M}ethods[\![\mathcal{G}et[\![\|\gamma\|^{\vec{l}}, \iota, l]\!]]\!]$		
$E^l[\mathbf{error}_j^k]$	$\longmapsto \mathbf{error}_j^k$	

Fig. 16. Reduction semantics for annotated CFCC

program components. It enables a formal examination of the behavior of the contract system independently of contract monitors and their label shuffling.

Before delving into the semantics of ownership propagation, we first introduce some useful notation for ownership labels handling. As values pass from one component to another during evaluation, they collect stacks of ownership annotations:

$$\begin{aligned} \|e\|^{\vec{l_n}} &= \|\ldots\|e\|^{l_1}\ldots\|^{l_n} \\ \|e\|^{\overleftarrow{l_n}} &= \|\ldots\|e\|^{l_n}\ldots\|^{l_1} \end{aligned}$$

We drop the subscript n when the height of the stack is irrelevant. Furthermore, we define the following operations on ownership stacks, which we use in the ownership propagating revision of the meta-functions:

$$\begin{aligned} k \otimes \vec{l_n} &= \begin{cases} l_1 & \text{if } \vec{l_n} = l_1 l_2 \ldots l_n \text{ where } n \neq 0 \\ k & \text{otherwise} \end{cases} \\ k \oplus \vec{l_n} &= \begin{cases} \vec{l_n} & \text{if } \vec{l_n} = l_1 l_2 \ldots l_n \text{ where } n \neq 0 \\ k & \text{otherwise} \end{cases} \end{aligned}$$

Like for CFCC without annotations, we divide the reduction rules into three groups: declarative rules, contract monitor rules and object-oriented rules. The rules from the first and second group are a superset of the rules for CPCF with annotations [Dimoulas et al. 2011]; the one extra rule is the one concerning class contracts monitors.

The most important rule in these two groups is the rule for function application. The β_v reduction — and its method-based relatives — 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:

$$E^l[\|\lambda(x_1\ldots x_n).e\|^{\vec{l}}\ v_1\ldots v_n] \longmapsto E^l[\|\{\|v_1\|^{l\overleftarrow{l}}\ldots\|v_n\|^{l\overleftarrow{l}}/x_1\ldots x_n\}e\|^{\vec{l}}]$$

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 $\vec{l}$. 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.

Rules that create new values, such as those for primitives, do not add an ownership annotation around their result. Instead, they let the context absorb the result.

Rules for monitors do not manipulate ownership at all because these annotations are merely proof tools. Thus when the rules decompose a contract monitor into its monitors for its sub-contracts or when the rules discharge a monitor, these action should happen in accordance with the component boundaries that ownership annotations specify and should not affect the flow of values between components or any ownership status. To that end, we leave ownership annotations and monitors independent from each other and then we demonstrate that they agree.

Object-oriented rules seem to have the least changes under the annotated semantics. However, the involvement of ownership is hidden in the definitions of the meta-functions. Figure 17 summarizes the modifications in the definitions of the meta-functions. Here we discuss and provide the full definitions of the major meta-functions. The rest can be found in appendix B.

Major Meta-functions	
$\mathcal{P}ull$	$\gamma \times m \rightarrow e$ same purpose as before except that along with contracts it collects ownership annotations
$\mathcal{P}ush$	$\gamma \times e \times \iota \times m \times l \rightarrow e$ same purpose as before except that along with contracts it collects ownership annotations; the l argument serves as an accumulator for keeping track of the owner of the level of class hierarchy the meta-function explores
$\mathcal{F}ind$	$\gamma \times \iota \times m \times l \rightarrow e$ same purpose as before; the l argument plays the same role as in $\mathcal{P}ush$
Helper Meta-functions	
$\mathcal{CV}$	$v \rightarrow \mathcal{T}rue$ or $\mathcal{F}alse$ same purpose as before
$\mathcal{M}ethods$	$\gamma \rightarrow \{m\}$ same purpose as before
$\{-/^{id}-\}-$	$\iota \times \iota \times e \rightarrow e$ same purpose as before
$\mathcal{O}bject$	$v \rightarrow \mathcal{T}rue$ or $\mathcal{F}alse$ same purpose as before
$\mathcal{I}s$	$\gamma \times \iota \rightarrow \mathcal{T}rue$ or $\mathcal{F}alse$ same purpose as before
$\mathcal{G}etS$	$\gamma \times \iota \times l \rightarrow cv$ same purpose as before; returns the super class with its owner's annotation; the l argument plays the same role as in $\mathcal{P}ush$
$\mathcal{G}et$	$\gamma \times \iota \times l \rightarrow cv$ same purpose as before; returns the class with its owner's annotation; the l argument plays the same role as in $\mathcal{P}ush$

Fig. 17. Meta-functions summary revisited

As terms are pulled or pushed, they pick up any ownership tags they meet. For instance in the definition of $\mathcal{P}ull$ in figure 18, the method implementation records the migration path of its journey. It collects any ownership annotations it runs into until it surfaces in the calling context. This way it manages to stay well-formed.

For the definitions of $\mathcal{F}ind$ in figure 19, we use an accumulator to keep track of the migration of method implementations. As $\mathcal{F}ind$ ascends the class hierarchy and runs 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. When it locates the superclass, it installs the method back in the calling context. To make sure that the method has the ownership annotation from the level of the class hierarchy from which it is retrieved, $\mathcal{F}ind$ has an argument that accumulates the owners of the class tables. When the meta-function moves to the next level in the class hierarchy, it updates the accumulator. Upon retrieval of the method implementation, the meta-function tags it with the owner label before passing it to the superclass. Since the method is well-formed, the ownership annotation ensures that it remains so in any context. The extra label argument serves a similar purpose for $\mathcal{G}etS$ (appendix B.6) and $\mathcal{G}et$ (appendix B.7). Finally, the definition of $\mathcal{P}ull$ in figure 20 employs an accumulator to keep track of the owner of the current recursion level. This accumulator does not play any role in ownership propagation but it is only useful for proving that the implementation remains well-formed as it ascends the class hierarchy.

$$
\begin{array}{lcl}
\mathcal{P}ull[\![\|\gamma\|^{\vec{k}}, m]\!] & = & \|\mathcal{P}ull[\![\|\gamma'\|^{\vec{l}}, m]\!]\|^{\vec{k}} \\
 & & \textbf{if}\, m \notin \{m_1, \ldots\} \\
 & & \textbf{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \\
\mathcal{P}ull[\![\|\gamma\|^{\vec{k}}, m]\!] & = & \|\lambda(this\ x_1 \ldots).e\|^{\vec{k}} \\
 & & \textbf{if}\, [m(this\ x_1 \ldots)\ e] \in \{[m_1(this_1\ x_1^1 \ldots)\ e_1], \ldots\} \\
 & & \textbf{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \\
\mathcal{P}ull[\![\|\textbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}, m]\!] & = & \|\mathcal{P}ull[\![\|\gamma\|^{\vec{l}}, m]\!]\|^{\vec{k}} \\
 & & \\
\mathcal{P}ull[\![\|\gamma\|^{\vec{k}}, m]\!] & = & \|\mathcal{P}ull[\![\|\gamma'\|^{\vec{l}}, m]\!]\|^{\vec{k}} \\
 & & \textbf{if}\, m \notin \{m_{p_1}, \ldots\} \\
 & & \textbf{where}\ \gamma = \mathbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad\qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
 & & \\
\mathcal{P}ull[\![\|\gamma\|^{\vec{k}}, m]\!] & = & \|\textbf{mon}_j^{k,l}(\kappa_{this}\ \kappa_1 \ldots \mapsto \kappa_r, \mathcal{P}ull[\![\|\gamma'\|^{\vec{l}}, m]\!])\|^{\vec{k}} \\
 & & \textbf{if}\ [m\ (\kappa_{this}\ \kappa_1 \ldots \mapsto \kappa_r)_j^{k,l}] \in \\
 & & \qquad\qquad \{[m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}], \ldots\} \\
 & & \textbf{where}\, \gamma = \mathbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad\qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

Fig. 18. The $\mathcal{P}ull$ meta-function revisited

The addition of ownership and obligations annotations slightly changes the appearance of our running example from section 4:

$$
\begin{array}{lcl}
d4 & = & \textbf{new}(loose\%) \\
loose\% & = & \textbf{class}\{\ resizable\% \\
 & & \quad \textbf{inherit}\ undersize? \\
 & & \quad \textbf{override}\ resize\ (this\ w\ h)\ ...\textbf{isend}^{\ddagger}(this, undersize?, w, h)...\} \\
resizable\% & = & checked{-}resizable\ dialog\% \\
checked\text{-}resizable & = & \textbf{mon}_j^{k,l_o}(make\text{-}resizable/c, \|make\text{-}resizable\|^k) \\
make\text{-}resizable/c & = & dom \mapsto rng \\
make\text{-}resizable & = & \lambda(base).\textbf{class}\{\ base \\
 & & \qquad\qquad \textbf{inherit}\ undersize? \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\quad resize\ (this\ w\ h)\ ... \\
 & & \qquad\qquad\quad halve\ (this)\ ...\textbf{isend}^{\ddagger}(this, resize, w, h)...\ \} \\
dom & = & \textbf{class/c}\{\ \textbf{public}\ undersize?\ (... \mapsto ...)\} \\
rng & = & \textbf{class/c}\{\ \textbf{public}\ resize\ (... \mapsto ...) \\
 & & \qquad\qquad \textbf{override}\ resize\ (... \mapsto ...)\}
\end{array}
$$

$$\begin{array}{ll}
\mathcal{F}ind[\![\|\gamma\|^{\vec{k}}, \iota, m, l]\!] & = \mathcal{P}ull[\![\|\gamma\|^{l \oplus \vec{k}}, m]\!] \\
& \quad \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
& \qquad\qquad \textbf{methods} \\
& \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\}
\end{array}$$

$$\begin{array}{ll}
\mathcal{F}ind[\![\|\gamma\|^{\vec{k}}, \iota, m, l]\!] & = \mathcal{P}ush[\![\|\gamma'\|^{\vec{l}}, \|\lambda(this\ x_1 \ldots).e\|^{(l \otimes \vec{k})^{\leftarrow}_{\vec{l}}}, \iota, m, l \otimes \vec{k}\ ]\!] \\
& \quad \text{if } [m(this\ x_1 \ldots)\ e] \in \{[m_1(this_1\ x_1^1 \ldots)\ e_1], \ldots\} \\
& \quad \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \|\gamma'\|^{\vec{l}} \\
& \qquad\qquad \textbf{methods} \\
& \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\}
\end{array}$$

$$\begin{array}{ll}
\mathcal{F}ind[\![\|\gamma\|^{\vec{k}}, \iota, m, l]\!] & = \mathcal{F}ind[\![\|\gamma'\|^{\vec{l}}, \iota, m, l \otimes \vec{k}\ ]\!] \\
& \quad \text{if } m \notin \{m_1, \ldots\} \\
& \quad \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \|\gamma'\|^{\vec{l}} \\
& \qquad\qquad \textbf{methods} \\
& \qquad\qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\}
\end{array}$$

$$\mathcal{F}ind[\![\|\textbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}, \iota, m, l]\!] = \mathcal{F}ind[\![\|\gamma\|^{\vec{l}}, \iota, m, l \otimes \vec{k}\ ]\!]$$

$$\begin{array}{ll}
\mathcal{F}ind[\![\|\gamma\|^{\vec{k}}, \iota, m, l]\!] & = \mathcal{F}ind[\![\|\gamma'\|^{\vec{l}}, \iota, m, l \otimes \vec{k}\ ]\!] \\
& \quad \text{where } \gamma = \mathbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
& \qquad\qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
& \qquad\qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}$$

Fig. 19. The $\mathcal{F}ind$ meta-function revisited

Evaluation in the annotated example proceeds the same way as in the example without annotations. Nevertheless, now the reductions and the meta-functions propagate the extra ownership information. From this perspective the first important reduction is the mixin application that results in $loose\%$. The application embeds the contract protected $dialog\%$ in the body of the mixin. Ownership annotations mark this value migration:

$$\begin{array}{l}
\textbf{mon}_j^{k,l_o}(rng, \|\textbf{class/v}^{\iota}\{\ \textbf{mon}_j^{l_o,k}(dom, \|dialog\%\|^{l_o}) \\
\qquad\qquad \textbf{methods} \\
\qquad\qquad resize\ (this\ w\ h)\ ... \\
\qquad\qquad halve\ (this)\ ...\textbf{isend}^{\iota}(this, resize, w, h)...\ \}\|^{k})
\end{array}$$

The program owner l_o owns the contracted term. However, the result of the mixin itself is property of the owner of the mixin k. Finally the superclass $dialog\%$ comes with the ownership tag for l_o, the whole program, which provides the class as an argument to the mixin.

In general, method calls and especially their use of $\mathcal{P}ull$, $\mathcal{F}ind$ and $\mathcal{P}ush$ are the interesting pieces with respect to ownership. For instance, the call $\|\textbf{send}(d4, halve, \|)^{l_o}$ uses $\mathcal{P}ull$ to locate the implementation of $halve$. The implementation of the method

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\|\gamma\|^{\vec{k}}, e, \iota, m, l]\!] & = & e \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\|\gamma\|^{\vec{k}}, e, \iota, m, l]\!] & = & \mathcal{P}ush[\![\|\gamma'\|^{\vec{l}}, \|e\|^{\overleftarrow{l}}, \iota, m, l \otimes \vec{k}\ ]\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\|\gamma\|^{\vec{k}}, e, \iota, m, l]\!] & = & \mathcal{P}ush[\![\|\gamma'\|^{\vec{l}}, \|e\|^{\overleftarrow{l}}, \iota, m, l \otimes \vec{k}\ ]\!] \\
 & & \text{if } m \notin \{m_{o_1}, \ldots\} \\
 & & \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}ush[\![\|\gamma\|^{\vec{k}}, e, \iota, m, l]\!] & = & \mathcal{P}ush[\![\|\gamma'\|^{\vec{l}}, \|\textbf{mon}_j^{k,l}(\kappa_{this}\ \kappa_1 \ldots \mapsto \kappa_r, e)\|^{\overleftarrow{l}}, \iota, m, l \otimes \vec{k}\ ]\!] \\
 & & \text{if } [m\ (\kappa_{this}\ \kappa_1 \ldots \mapsto \kappa_r)_j^{k,l}] \in \\
 & & \qquad \{[m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{k,l}], \ldots\} \\
 & & \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{l,k}] \ldots \\
 & & \qquad \textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{k,l}] \ldots\}
\end{array}
$$

Fig. 20. The $\mathcal{P}ush$ meta-function revisited

resides in $loose\%$ that has the same owner l_o as the caller object $d4$ and the call site. Thus, $\mathcal{P}ull$ does not decorate the implementation with any ownership annotations. However, things are different when the evaluation trace reaches the local method call $\textbf{isend}^{\iota'}(\|d4\|^{kl_o}, resize, w, h)$. First we observe that due to the call to $halve$ the object $d4$ migrates inside the class where the local method call resides. The β-reduction makes sure that this migration is recorded in the ownership annotation of the object. Second, as we have already explained in section 4, this local method call concerns an overridden method. Thus, $\mathcal{F}ind$ delegates the method implementation contract wrapping to $\mathcal{P}ush$. To help $\mathcal{P}ush$ attach the right ownership information to the method implementation, $\mathcal{F}ind$ uses its accumulator to keep track of the owner of the class hierarchy level it traverses. Since $loose\%$ contains the method implementation and $loose\%$ belongs to l_o, $\mathcal{F}ind$ passes the method implementation, $\lambda(this\ w\ h)....$, to $\mathcal{P}ush$ as $\|\lambda(this\ w\ h)....\|^{l_o}$. $\mathcal{P}ush$ guides the implementation deeper into the class hierarchy collecting contracts and ownership annotations. Finally, the term that emerges at the call site has the form $\|\textbf{mon}_j^{l_o,k}(... \mapsto ..., \|\lambda(this\ w\ h)....\|^{l_o})\|^{k}$. Notice that ownership gives a flow-oriented justification for the label flipping for override contracts. The meta-function collects override contracts while ascending the class hierarchy as it moves the implementation from subclasses to superclasses. Thus the subclasses play the role of the server for these contracts and the superclasses the role of the client. This is the opposite order compared

to how the contracts are attached to classes, so switching the labels is necessary for correct blame assignment.

In summary, 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 machinery to keep track of the flow of inherited or overridden methods. As the definition of our meta-functions demonstrate, however, this machinery can be set up in an 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 *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).

Note: CFCC from section 4 and CFCC with annotations from this section are equivalent. The annotations do not interfere or affect computation in any way. We prove this conjecture by relating plain terms and contracts to their annotated counterparts. We define this relation as the compatible closure, $\simeq_{ctx}$, of the following two rules:

$$\frac{}{e \simeq_{ctx} \|e\|^{l}} \qquad \frac{e \simeq e'}{\mathbf{flat}(e) \simeq \lfloor \mathbf{flat}(e) \rfloor^{\bar{l}}}$$

We show, then, that $\simeq_{ctx}$, is a lock-step bisimulation between non-annotated and annotated terms.

PROPOSITION 5.2. *If* $e \simeq e'$ *then* $e \overset{*}{\hookrightarrow} v$ *iff* $e' \overset{*}{\longmapsto} v'$ *and* $v \simeq v'$.

In short, we establish that the two languages define the same behavior for related programs.

6. BLAME CORRECT: THE PROOF

With ownership and obligation annotations we can formulate our blame correctness criterion.

Definition 6.1 (*Blame Correctness*). A contract system is *blame correct* if for all terms e_0 such that $l_o \vdash e_0$ and $e_0 \overset{*}{\longmapsto} E^{\dagger}[\mathbf{mon}_{\dagger}^{k,\dagger}(\lfloor \mathbf{flat}(e_1) \rfloor^{\bar{l}}, v_1)]$, we have $v_1 = \|v\|^{k}$ and $k \in \bar{l}$. Note: The identity of the † 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 k on the monitor is the owner of the value and the flat contract is the server's obligation. 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 5.1 into a logical statement about reduction systems.

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 $k; l; j \rhd \kappa$ imply the desired property for monitors of flat contracts. Unfortunately, the reduction process generates expressions that do not preserve these judgments. Fortunately, 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-formedness 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 reduction that is responsible for deviating from well-formed terms is the evaluation of function contract monitors:

$$E^l[\mathbf{mon}_j^{k,l}(\kappa_1 \dots \kappa_n \mapsto \kappa_r, v)] \\ \longmapsto E^l[\lambda(x_1 \dots x_n).\mathbf{mon}_j^{k,l}(\kappa_r, v\ \mathbf{mon}_j^{l,k}(\kappa_1, x_1) \dots \mathbf{mon}_j^{l,k}(\kappa_n, x_n))]$$

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.

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 $k; l; j; \Gamma \rhd \kappa$.[5] 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 figures 21 and 22, we provide the rules of the generalized judgment for well-formed terms and contracts.

According to figure 21, the rule for $\mathbf{mon}_j^{k,l}(\kappa, e)$ 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 (figure 23).

Finally, we add an extra rule for terms with ownership annotations, $\|e\|^k$. These terms mark an ownership change and thus they are well-formed iff e is well-formed under k. This rule is not necessary for source code as we restrict ownership annotations to monitors. However, as the evaluation proceeds ownership annotations may show up in other places too and so we need rules for these cases. Before we move to the proof that CFCC is blame correct, we first show that the new judgment for well-formed terms generalizes the initial one.

PROPOSITION 6.2. *If $l \vdash e$ then $l; \varnothing \vdash e$.*

PROOF. We proceed by induction on the height of the derivation of $l \vdash e$. ∎

Now, with $l; \Gamma \vdash e$, we can prove the central subject reduction lemma of the blame correctness proof.

MAIN LEMMA 6.3. *If $l_o; \varnothing \vdash e$ and $e \longmapsto e_0$, then $l_o; \varnothing \vdash e_0$.*

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].

— $E^l[\|n_1\|^{\vec{k}} + \|n_2\|^{\vec{l}}] \longmapsto E^l[n]$

By assumption $l_o; \varnothing \vdash e$, for which lemma 6.5 implies that $l; \varnothing \vdash \|n_1\|^{\vec{k}} + \|n_2\|^{\vec{l}}$. We can use the same label to check n, i.e., $l; \varnothing \vdash n$. Hence, $l_o; \varnothing \vdash E^l[n]$.

— The cases for other primitive operations are similar to the first.

[5]Following the type systems tradition, we could have added the environment as early as in the rules for well-formed source programs. However variables are not a problem at that level as, even when they show up in monitors, they come with the appropriate ownership annotations due to well-formedness. We have therefore delayed the introduction of the environment until it is really necessary.

$\boxed{l;\Gamma \vdash e}$

$$\frac{l;\Gamma \vdash e_f \quad l;\Gamma \vdash e_1 \quad \ldots}{l;\Gamma \vdash e_f\ e_1 \ldots} \qquad \frac{}{l;\Gamma \vdash \mathbf{error}^k} \qquad \frac{l;\Gamma \vdash e_1}{l;\Gamma \vdash \mathbf{zero?}(e_1)}$$

$$\frac{l;\Gamma \vdash e_1 \quad l;\Gamma \vdash e_2 \quad l;\Gamma \vdash e_3}{l;\Gamma \vdash \mathbf{if}\ e_1\ e_2\ e_3} \qquad \frac{l;\Gamma \vdash e_1 \quad l;\Gamma \vdash e_2}{l;\Gamma \vdash e_1 + e_2} \qquad \frac{l;\Gamma \vdash e_1 \quad l;\Gamma \vdash e_2}{l;\Gamma \vdash e_1 - e_2}$$

$$\frac{l;\Gamma \vdash e_1 \quad l;\Gamma \vdash e_2}{l;\Gamma \vdash e_1 \wedge e_2} \qquad \frac{l;\Gamma \vdash e_1 \quad l;\Gamma \vdash e_2}{l;\Gamma \vdash e_1 \vee e_2} \qquad \frac{}{l;\Gamma \vdash \mathbf{tt}} \qquad \frac{}{l;\Gamma \vdash \mathbf{ff}}$$

$$\frac{}{l;\Gamma \vdash x} \qquad \frac{}{l;\Gamma \vdash n} \qquad \frac{l;\Gamma \uplus \{x_1 : l\} \uplus \ldots \vdash e}{l;\Gamma \vdash \lambda(x_1 \ldots).e} \qquad \frac{k;\Gamma \Vdash e \quad k;l;j;\Gamma \rhd \kappa}{l;\Gamma \vdash \mathbf{mon}_j^{k,l}(\kappa, e)}$$

$$\frac{k;\Gamma \vdash e}{l;\Gamma \vdash \|e\|^l} \qquad \frac{}{l;\Gamma \vdash \mathbf{object\%}} \qquad \frac{l;\Gamma \vdash e}{l;\Gamma \vdash \mathbf{object}(e)} \qquad \frac{l;\Gamma \vdash e}{l;\Gamma \vdash \mathbf{new}(e)}$$

$$\frac{l;\Gamma \vdash e_o \quad l;\Gamma \vdash e_1 \quad \ldots}{l;\Gamma \vdash \mathbf{send}(e_o, m, e_1 \ldots)}$$

$$\frac{l;\Gamma \vdash e_o \quad l;\Gamma \vdash e_1 \quad \ldots}{l;\Gamma \vdash \mathbf{isend}^\iota(e_o, m, e_1 \ldots)} \qquad \frac{l;\Gamma \vdash e_o \quad l;\Gamma \vdash e_1 \quad \ldots}{l;\Gamma \vdash \mathbf{super}^\iota(e_o, m, e_1 \ldots)}$$

$$\frac{\begin{array}{c} l;\Gamma \vdash e_s \\ l;\Gamma \uplus \{this_{p_1} : l\} \uplus \{x_1^{p_1} : l\} \uplus \ldots \vdash e_{p_1} \quad \ldots \\ l;\Gamma \uplus \{this_{o_1} : l\} \uplus \{x_1^{o_1} : l\} \uplus \ldots \vdash e_{o_1} \quad \ldots \end{array}}{\begin{array}{l} l;\Gamma \vdash \mathbf{class}\{\ e_s \\ \qquad \mathbf{inherit}\ m_{i_1} \ldots \\ \qquad \mathbf{public}\ [m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ e_{p_1}] \ldots \\ \qquad \mathbf{override}\ [m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ e_{o_1}] \ldots \} \end{array}}$$

$$\frac{\begin{array}{c} k;\Gamma \vdash v \\ k;l;j;\Gamma \rhd \kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1} \quad \ldots \\ l;k;j;\Gamma \rhd \kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1} \quad \ldots \end{array}}{\begin{array}{l} l;\Gamma \vdash \mathbf{G}\{\ \|v\|^k \\ \qquad \mathbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\ \qquad \mathbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots \} \end{array}}$$

$$\frac{l;\Gamma \vdash \|\gamma\|^{\vec{l}} \quad l;\Gamma \uplus \{this_1 : l\} \uplus \{x_1^1 : l\} \uplus \ldots \vdash e_1 \quad \ldots}{l;\Gamma \vdash \mathbf{class/v}^\iota\{\ \|\gamma\|^{\vec{l}}\ \mathbf{methods}\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots \}}$$

Fig. 21. Ownership coincides with contract monitors (2)

$\boxed{k; l; j; \Gamma \rhd \kappa}$

$$\frac{l; k; j; \Gamma \rhd \kappa_1 \quad \ldots \quad l; k; j; \Gamma \rhd \kappa_n \quad k; l; j; \Gamma \rhd \kappa_r}{k; l; j; \Gamma \rhd \kappa_1 \ldots \kappa_n \mapsto \kappa_r}$$

$$\frac{j; \Gamma \vdash e}{k; l; j; \Gamma \rhd \lfloor \mathbf{flat}(\|e\|^j) \rfloor^k}$$

$$\frac{\begin{array}{ll} k; l; j; \Gamma \rhd \kappa^{p_1}_{this}\ \kappa^{p_1}_1 \ldots \mapsto \kappa^{p_1}_r & \ldots \\ l; k; j; \Gamma \rhd \kappa^{o_1}_{this}\ \kappa^{o_1}_1 \ldots \mapsto \kappa^{o_1}_r & \ldots \end{array}}{\begin{array}{ll} k; l; j; \Gamma \rhd & \mathbf{class/c}\{ \\ & \mathbf{public} \\ & \quad [m_{p_1}\ (\kappa^{p_1}_{this}\ \kappa^{p_1}_1 \ldots \mapsto \kappa^{p_1}_r)] \ldots \\ & \mathbf{override} \\ & \quad [m_{o_1}\ (\kappa^{o_1}_{this}\ \kappa^{o_1}_1 \ldots \mapsto \kappa^{p_1}_r)] \ldots \} \end{array}}$$

Fig. 22. Obligations coincide with labels on monitors (2)

$\boxed{l; \Gamma \Vdash e}$

$$\frac{k; \Gamma \vdash e}{l; \Gamma \Vdash \|e\|^k} \qquad \frac{\Gamma(x) = l}{l; \Gamma \Vdash x} \qquad \frac{l; \Gamma \vdash e_f \quad l; \Gamma \vdash e_1 \quad \ldots}{l; \Gamma \Vdash \|e_f\|^l e_1 \ldots}$$

Fig. 23. Ownership is preserved for monitored expressions

— $E^l[\|\lambda(x_1 \ldots x_n).e_0\|^{\vec{k}}\ v_1 \ldots v_n] \longmapsto E^l[\|\{\|v_1\|^{l^{\overleftarrow{k}}} \ldots \|v_n\|^{l^{\overleftarrow{k}}}/x_1 \ldots x_n\}e_0\|^{\vec{k}}]$
Again by assumption and lemma 6.5, we conclude that $l; \varnothing \vdash \|\lambda(x).e_0\|^{\vec{k}}\ v_1 \ldots v_n$ and, therefore, we deduce that $l; \varnothing \vdash \|\lambda(x_1 \ldots x_n).e_0\|^{\vec{k}}$ and for all i, $l; \varnothing \vdash v_i$.
Next we distinguish two cases, depending on the length of $\vec{k}$. First assume the vector is empty. In that case, the rules imply $l; \{x_1 : l\} \uplus \ldots \uplus \{x_n : l\} \vdash e_0$. Combining this judgment with $l; \varnothing \vdash v$, we may conclude that $l; \varnothing \vdash \{\|v_1\|^l \ldots \|v_n\|^l/x_1 \ldots x_n\}e_0$ via lemma 6.6. Finally from here it is easy to get the desired conclusion, $l_o; \varnothing \vdash E^l[\{\|v_1 \ldots v_n\|^l/x_1 \ldots x_n\}e_0]$. Second, let k_1 be the first element of $\vec{k}$. In that case, the rules imply $k_1; \{x_1 : k_1\} \uplus \ldots \uplus \{x_n : k_1\} \vdash e_0$. Since $l; \varnothing \vdash v$ still holds, we conclude again via lemma 6.6 that $k_1; \varnothing \vdash \{\|v_1\|^{l^{\overleftarrow{k}}} \ldots \|v_n\|^{l^{\overleftarrow{k}}}/x_1 \ldots x_n\}e_0$. Since k_1 is the outermost element of $\vec{k}$, we finally get the desired result, $l_o; \varnothing \vdash E^l[\|\{\|v_1\|^{l^{\overleftarrow{k}}} \ldots \|v_n\|^{l^{\overleftarrow{k}}}/x_1 \ldots x_n\}e_0\|^{\vec{k}}]$.
— $E^l[\mathbf{new}(v)] \longmapsto E^l[\mathbf{object}(v)]$
By assumption $l_o; \varnothing \vdash e$, for which lemma 6.5 implies that $l; \varnothing \vdash \mathbf{new}(v)$. The rules imply $l; \varnothing \vdash v$. We can use the same label to establish that $l; \varnothing \vdash \mathbf{object}(v)$. Hence, $l_o; \varnothing \vdash E^l[\mathbf{object}(v)]$.
— $E^l[c] \longmapsto E^l[cv]$
where $c = \mathbf{class}\{\ v_s$
$\mathbf{inherit}\ m_{i_1} \ldots$
$\mathbf{public}\ [m_{p_1}(this_{p_1}\ x^{p_1}_1 \ldots)\ e_{p_1}] \ldots$
$\mathbf{override}\ [m_{o_1}(this_{o_1}\ x^{o_1}_1 \ldots)\ e_{o_1}] \ldots\}$

and $cv = \textbf{class/v}^{\iota}\{\ v_s$
$\quad\textbf{methods}$
$\quad[m_{p_1}(this_{p_1}\ x_1^{p_1}\ldots)\ \{\iota/^{id}\ddagger\}e_{p_1}]\ldots$
$\quad[m_{o_1}(this_{o_1}\ x_1^{o_1}\ldots)\ \{\iota/^{id}\ddagger\}e_{o_1}]\ldots\ \}$

Using the assumptions and lemma 6.5, we get $l;\varnothing\vdash c$. Thus, by the rules, $l;\varnothing\vdash v_s$, $l;\{this_{p_1}:l,x_1^{p_1}:l,\ldots\}\vdash e_{p_1}$, ... and $l;\{this_{o_1}:l,x_1^{o_1}:l,\ldots\}\vdash e_{o_1}$, ... By lemma 6.8, we get $l;\{this_{p_1}:l,x_1^{p_1}:l,\ldots\}\vdash\{\iota/^{id}\ddagger\}e_{p_1}$, ... and $l;\{this_{o_1}:l,x_1^{o_1}:l,\ldots\}\vdash\{\iota/^{id}\ddagger\}e_{o_1}$, ... With an application of the rules for well-formed expressions, we obtain $l;\varnothing\vdash cv$ and $l_o;\varnothing\vdash E^l[cv]$.

— $E^l[\textbf{send}(v_o,m,v\ldots)]\longmapsto E^l[\mathcal{P}ull[\![v_o,m]\!]\ v_o\ v_1\ldots v_n]$
By the assumptions and lemma 6.5, we conclude $l;\varnothing\vdash\textbf{send}(v_o,m,v\ldots)$. Thus, by the rules, $l;\varnothing\vdash v_1$, ..., $l;\varnothing\vdash v_n$, and $l;\varnothing\vdash v_o$ and by lemma 6.9, $l;\varnothing\vdash\mathcal{P}ull[\![v_o,m]\!]$. Finally by the rules, we get $l;\varnothing\vdash\mathcal{P}ull[\![v_o,m]\!]\ v_o\ v_1\ldots v_n$ and we obtain $l_o;\varnothing\vdash E^l[\mathcal{P}ull[\![v_o,m]\!]\ v_o\ v_1\ldots v_n]$.

— $E^l[\textbf{super}^{\iota}(v_o,m,v_1\ldots v_n)]\longmapsto E^l[\mathcal{P}ull[\![\mathcal{G}etS[\![v_o,\iota,l]\!],m]\!]\ v_o\ v_1\ldots v_n]$
By the usual argument, we obtain $l;\varnothing\vdash v_1$, ..., $l;\varnothing\vdash v_n$, and $l;\varnothing\vdash v_o$. Lemmas 6.9 and 6.10 give us $l;\varnothing\vdash\mathcal{P}ull[\![\mathcal{G}etS[\![v_o,\iota,l]\!],m]\!]$. Applying the rules for well-formed expressions, we conclude $l;\varnothing\vdash\mathcal{P}ull[\![\mathcal{G}etS[\![v_o,\iota,l]\!],m]\!]\ v_o\ v_1\ldots v_n$ and $l_o;\varnothing\vdash E^l[\mathcal{P}ull[\![\mathcal{G}etS[\![v_o,\iota,l]\!],m]\!]\ v_o\ v_1\ldots v_n]$.

— $E^l[\textbf{isend}^{\iota}(v_o,m,v_1\ldots v_n)]\longmapsto E^l[\mathcal{F}ind[\![v_o,\iota,m,l]\!]\ v_o\ v_1\ldots v_n]$
As in the previous cases, we get the following: $l;\varnothing\vdash v_1$, ..., $l;\varnothing\vdash v_n$, and $l;\varnothing\vdash v_o$. By lemma 6.12, we obtain $l;\varnothing\vdash\mathcal{F}ind[\![v_o,\iota,m,l]\!]$. We put all the pieces together with the rules for well-formed expressions to get $l;\varnothing\vdash\mathcal{F}ind[\![v_o,\iota,m,l]\!]\ v_o\ v_1\ldots v_n$ and finally, $l_o;\varnothing\vdash E^l[\mathcal{F}ind[\![v_o,\iota,m,l]\!]\ v_o\ v_1\ldots v_n]$.

— $E^l[\textbf{mon}_j^{k,l}(\lfloor\textbf{flat}(e_c)\rfloor^{\bar{l}},v)]\longmapsto E^l[\textbf{if}\ (e_c\ v)\ v\ \textbf{error}^k]$
The assumptions imply $l;\varnothing\vdash\textbf{mon}_j^{k,l}(\lfloor\textbf{flat}(e_c)\rfloor^{\bar{l}},v)$ via lemma 6.5 and hence, $e_c=\|e_c'\|^j$ with $j;\varnothing\vdash e_c'$. Furthermore, the same reasoning yields $v=\|v_0\|^k$ and $k;\varnothing\vdash v_0$. Since the rules for well-formed expressions imply $l;\varnothing\vdash\textbf{if}\ (e_c\ v)\ v\ \textbf{error}^k$ is well-formed, the desired conclusion follows immediately.

— $E^l[\textbf{mon}_j^{k,l}(\kappa_1\ldots\kappa_n\mapsto\kappa_r,v)]\longmapsto E^l[\lambda(x_1\ldots x_n).\textbf{mon}_j^{k,l}(\kappa_r,v\ \textbf{mon}_j^{l,k}(\kappa_1,x_1)\ldots)]$
With the usual reasoning, we conclude that $v=\|v_0\|^k$, $k;\varnothing\vdash v_0$, and $l;\varnothing\vdash\textbf{mon}_j^{k,l}(\kappa_1\ldots\kappa_n\mapsto\kappa_r,v)$. The contract check yields $n+1$ pieces of knowledge:

$$l;k;j;\varnothing\rhd\kappa_1$$

$$\ldots$$

$$l;k;j;\varnothing\rhd\kappa_n$$

and

$$k;l;j;\varnothing\rhd\kappa_r.$$

From an additional application of well-formedness we get $k;\{x_1:l\}\uplus\ldots\uplus\{x_n:l\}\Vdash v\ \textbf{mon}_j^{l,k}(\kappa_1,x_1)\ldots$ and, with lemma 6.7, we conclude the that $l;\{x_1:l\}\uplus\ldots\uplus\{x_n:l\}\vdash\textbf{mon}_j^{k,l}(\kappa_2,v\ \textbf{mon}_j^{l,k}(\kappa_1,x_1)\ldots)$. Finally from a last application of the rules for well-formedness we get $l;\varnothing\vdash\lambda(x_1\ldots x_n).\textbf{mon}_j^{k,l}(\kappa_2,v\ \textbf{mon}_j^{l,k}(\kappa_1,x)\ldots)$.

— $E^l[\textbf{mon}_j^{k,l}(\kappa,v)]\longmapsto E^l[v']$
where $\kappa=\ \textbf{class/c}\{\ \textbf{public}\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1}\ldots\mapsto\kappa_r^{p_1})]\ldots$
$\qquad\qquad\textbf{override}\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1}\ldots\mapsto\kappa_r^{o_1})]\ldots\}$
and

$v' = \mathbf{G}\{\ v$
public $[m_{p_1}\ (\kappa^{p_1}_{this}\ \kappa^{p_1}_1 \ldots \mapsto \kappa^{p_1}_r)^{k,l}_j] \ldots$
override $[m_{o_1}\ (\kappa^{o_1}_{this}\ \kappa^{o_1}_1 \ldots \mapsto \kappa^{o_1}_r)^{l,k}_j] \ldots\}$

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

$$l; k; j; \varnothing \rhd \kappa^{p_1}_{this}\ \kappa^{p_1}_1\ \ldots \mapsto \kappa^{p_1}_r \qquad \ldots$$

and

$$k; l; j; \varnothing \rhd \kappa^{o_1}_{this}\ \kappa^{o_1}_1\ \ldots \mapsto \kappa^{o_1}_r \qquad \ldots$$

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

— $E^l[\mathbf{mon}^{k,l}_j(\mathbf{any}, v)] \longmapsto E^l[v]$

As usual, we deduce that $v = \|v_0\|^k$, $k; \varnothing \vdash v_0$, and $l; \varnothing \vdash \mathbf{mon}^{k,l}_j(\mathbf{any}, v)$. 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 6.3 requires several lemmas regarding properties of the evaluation contexts, substitution and the meta-functions with respect to the judgment for well-formed terms. We list them and their central proof ideas.

LEMMA 6.4. *For all terms e there are unique evaluation context E^l and redex r such that $e = E^l[r]$ or $e = v$.*

PROOF. By induction on the structure of e. ∎

LEMMA 6.5. *If $l; \varnothing \vdash E^k[e]$ then $k; \varnothing \vdash e$.*

PROOF. By induction on the size of E^k. ∎

LEMMA 6.6. *If $l; \Gamma \uplus \{x_1 : k_1\} \uplus \ldots \uplus \{x_n : k_n\} \vdash e$, for all i, $v_i = \|v'_i\|^{k_i}$ and $k_i; \Gamma \vdash v'_i$ then*

$$l; \Gamma \vdash \{v_1 \ldots v_n / x_1 \ldots x_n\} e.$$

PROOF. By induction on the height of $l; \Gamma \uplus \{x_1 : k_1\} \uplus \ldots \uplus \{x_n : k_n\} \vdash e$. ∎

LEMMA 6.7. *If $l; \Gamma \vdash e$ and for all i, $x_i \notin dom\,(\Gamma)$, then*

$$l; \Gamma \uplus \{x_1 : k_1\} \uplus \ldots \uplus \{x_n : k_n\} \vdash e.$$

PROOF. By induction on the height of $l; \Gamma \vdash e$. ∎

LEMMA 6.8. *If $l; \Gamma \vdash e$, then $l; \Gamma \vdash \{\iota/^{id}\ddagger\} e$.*

PROOF. By induction on the height of $k; \Gamma \vdash e$. ∎

LEMMA 6.9. *If $k; \varnothing \vdash \|\gamma\|^{\vec{l}}$, $m \in \mathcal{M}ethods[\![\|\gamma\|^{\vec{l}}]\!]$, and $\mathcal{CV}[\![\|\gamma\|^{\vec{l}}]\!]$ or $\mathcal{O}bject[\![\|\gamma\|^{\vec{l}}]\!]$, then*

$$k; \varnothing \vdash \mathcal{P}ull[\![\|\gamma\|^{\vec{l}}, m]\!].$$

PROOF. By induction on the height of $k; \varnothing \vdash \|\gamma\|^{\vec{l}}$. ∎

LEMMA 6.10. *If $l; \varnothing \vdash \|\gamma\|^{\vec{l}}$, $\mathcal{I}s[\![\|\gamma\|^{\vec{l}}, \iota]\!]$, and $\mathcal{CV}[\![\|\gamma\|^{\vec{l}}]\!]$ or $\mathcal{O}bject[\![\|\gamma\|^{\vec{l}}]\!]$, then*

$$\text{for all } k,\ k; \varnothing \vdash \mathcal{G}etS[\![\|\gamma\|^{\vec{l}}, \iota, l]\!] \text{ and } \mathcal{CV}[\![\mathcal{G}etS[\![\|\gamma\|^{\vec{l}}, \iota, l]\!]]\!].$$

PROOF. By induction on the height of $l; \varnothing \vdash \|\gamma\|^{\vec{l}}$. ∎

LEMMA 6.11. *If* $l; \varnothing \vdash \|\gamma\|^{\vec{l}}$, $\mathcal{I}s[\![\|\gamma\|^{\vec{l}}, \iota]\!]$, $\mathcal{CV}[\![\|\gamma\|^{\vec{l}}]\!]$, $m \in \mathcal{M}ethods[\![\mathcal{G}et[\![\|\gamma\|^{\vec{l}}, \iota]\!]]\!]$, $v = \|v_o\|^{l \otimes \vec{l}}$, *and* $l \otimes \vec{l}; \varnothing \vdash v_o$, *then*

$$\textit{for all } k,\ k; \varnothing \vdash \mathcal{P}ush[\![\|\gamma\|^{\vec{l}}, v, \iota, m, l]\!].$$

PROOF. By induction on the height of $l; \varnothing \vdash \|\gamma\|^{\vec{l}}$. ∎

LEMMA 6.12. *If* $l; \varnothing \vdash \|\gamma\|^{\vec{l}}$, $m \in \mathcal{M}ethods[\![\mathcal{G}et[\![\|\gamma\|^{\vec{l}}, \iota]\!]]\!]$, $\mathcal{I}s[\![\|\gamma\|^{\vec{l}}, \iota]\!]$, *and* $\mathcal{CV}[\![\|\gamma\|^{\vec{l}}]\!]$ *or* $\mathcal{O}bject[\![\|\gamma\|^{\vec{l}}]\!]$, *then*

$$\textit{for all } k,\ k; \varnothing \vdash \mathcal{F}ind[\![\|\gamma\|^{\vec{l}}, \iota, m, l]\!].$$

PROOF. By induction on the height of $l; \varnothing \vdash \|\gamma\|^{\vec{l}}$ using lemmas 6.9 and 6.11 for the two base cases. ∎

The main lemma 6.3 is all we need to show that CFCC is blame correct.

THEOREM 6.13. $\longmapsto$ *is blame correct.*

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

$$l_o; \varnothing \vdash E^l[\mathbf{mon}_j^{k,l}(\lfloor \mathbf{flat}(e_c) \rfloor^k, \|v\|^k)]$$

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 6.5 the contract is part of the obligations of the server and carries the server's label. ∎

7. IMPLEMENTATION

While our formal model communicates the basic design and helps us prove basic properties, it does not 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 values with behavior, they return similar values restricted to good behavior from the client/server perspective.

Thus, **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 diagrams in figure 24. The first diagram presents a source configuration complete with classes, objects, contracts, and boundaries. In the second diagram, the dotted squares represent classes created from applying projections to classes. The inheritance arrows connect the derived classes to the originals.

This section consists of two parts. The first one explains how classes and objects are implemented and equipped with basic contracts. The second part revises this implementation for internal dynamic dispatches.

7.1. Basics

Both classes and objects are implemented [Flatt et al. 2006] as heterogeneous and opaque structures. Opaqueness guarantees safety. The structure contains all the necessary information for constructing objects and accessing methods and fields. Object

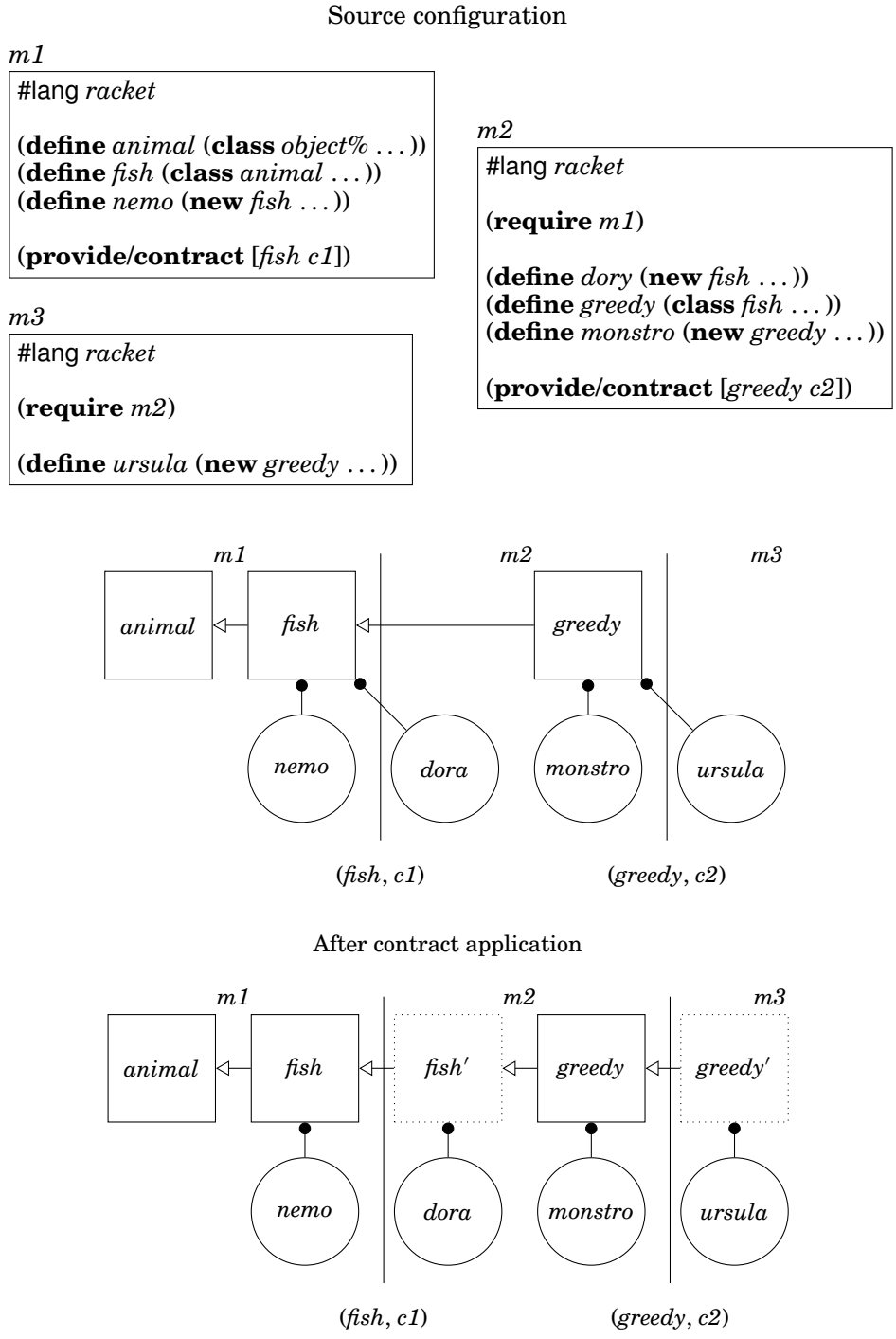


Fig. 24. Classes, objects, and contract boundaries

construction is handled by functions that create structural representations of objects using the initialization arguments and the initialization expressions of the class.

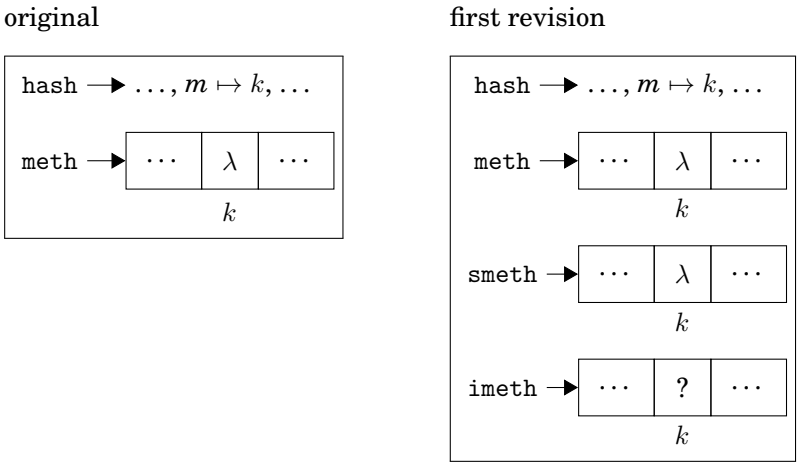


Fig. 25. Method table organization for contracts

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 local 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 index into the method table of the super class.

In short, all method calls access the same method vector, whether they are local method calls, method **send**s, or **super** calls. Each kind of contract clause affects a different kind of call, however. Applying contract projections can be a costly operation, so we do not store the different projections and apply the correct projection for a particular call. Instead, our implementation trades space for run-time speed; that is, it separates out the method vector into three vectors, illustrated by figure 25. Method sends via **send** use the `meth` vector, **super** calls access `smeth`, and local method calls go through `imeth`. The revised compiler converts each operation into an indexed access to the appropriate vector. This separation works for methods sends and super calls; local method calls require additional changes, which we describe in section 7.2.

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 appropriate pair of functions is replaced with a new pair. 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 passes the contracted arguments to the original initializer.

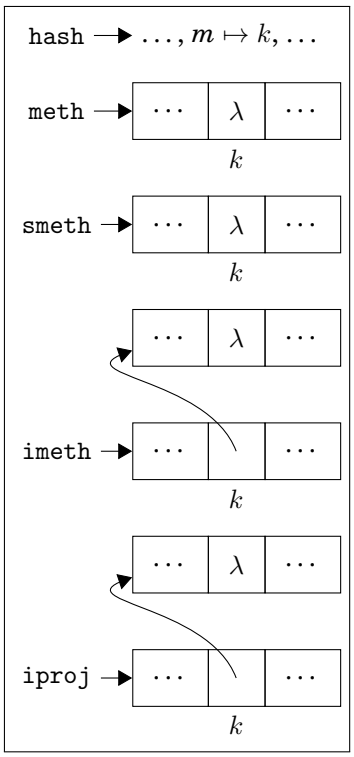


Fig. 26. Second revision of class method tables for contracts

7.2. 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 local method calls where each entry maps to a vector of method implementations; see figure 26. The separation of vector entries based on contract boundaries means that contract projections for local 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 26 shows that the class representation contains a table with **override** contracts labelled `iproj`. These are needed so that a future extension of the class can enforce the **override** contract clauses of this class.

Figure 27 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 local method 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 class with contracts applied. The different method implementations are differentiated using subscripts, and primed method names denote contracted methods.

The vectors for a method contains an entry for each contract boundary between its class and the first class that defined the 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

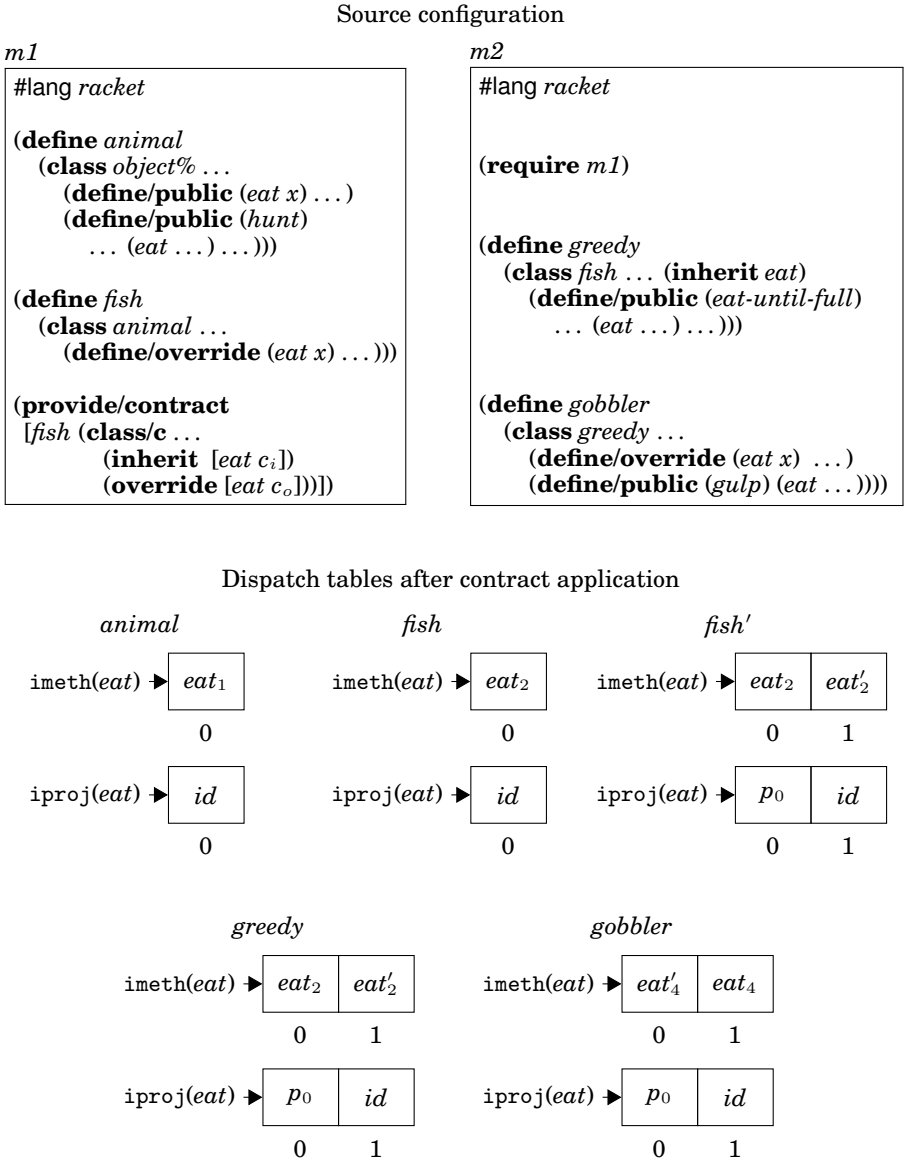


Fig. 27. Classes with **override** and **inherit** contracts

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 a local 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'_4. If the code calls *gulp*, the local method call from *greedy* uses index 1. Thus, it retrieves eat_4, 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 27.

If a subclass does not override a method, as in *greedy*, its method and projection vectors are those from its superclass. If it does, 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'_4, the result of composing eat_4 with the projection p_o, the implementation of contract c_o. The method at index 1 is just eat_4, 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'_2 at index 1 is the result of applying the contract projection for c_i (from the class contract for *fish*) to eat_2. The projection p_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.

8. EVALUATING CLASS CONTRACTS

Our next task is to show that our proposed language extension is both practical and requires an acceptable overhead. 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.

8.1. 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 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.

Table I 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.

Table I. Micro-benchmark results (times in milliseconds)

	rev	reps	min	avg	max	norm
External field access	port	$2.5 * 10^7$	1928	2016.88	2140	1
External field access	class/c	$2.5 * 10^7$	1696	1897.16	2248	0.94
Internal (same class) field access	port	$2.5 * 10^8$	3588	3665.84	4176	1
Internal (same class) field access	class/c	$2.5 * 10^8$	3584	3632.12	3692	0.99
Internal (super class) field access	port	$2.5 * 10^8$	3588	3665.60	4184	1
Internal (super class) field access	class/c	$2.5 * 10^8$	3584	3636.44	4252	0.99
External method access	port	$2.5 * 10^7$	1948	2076.28	3060	1
External method access	class/c	$2.5 * 10^7$	1964	2074.20	2196	1.00
Internal (same class) method access	port	$2.5 * 10^7$	852	889.72	936	1
Internal (same class) method access	class/c	$2.5 * 10^7$	916	950.56	1000	1.07
Internal (super class) method access	port	$2.5 * 10^7$	856	890.12	956	1
Internal (super class) method access	class/c	$2.5 * 10^7$	924	952.40	988	1.07
Object creation	port	$2.5 * 10^7$	5229	5457.52	6461	1
Object creation	class/c	$2.5 * 10^7$	5317	5488.08	7377	1.01
Subclassing (no methods/fields)	port	$2.5 * 10^5$	3832	4020.08	4328	1
Subclassing (no methods/fields)	class/c	$2.5 * 10^5$	3916	4077.16	4284	1.01
Subclassing (no fields)	port	$2.5 * 10^5$	4176	4377.04	4685	1
Subclassing (no fields)	class/c	$2.5 * 10^5$	4493	4650.04	4905	1.06
Subclassing (no methods)	port	$2.5 * 10^5$	4040	4230.04	4604	1
Subclassing (no methods)	class/c	$2.5 * 10^5$	4484	4660.04	4976	1.10
Subclassing (methods/fields)	port	$2.5 * 10^5$	4449	4644.40	5052	1
Subclassing (methods/fields)	class/c	$2.5 * 10^5$	4948	5141.12	5444	1.11

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.

In our first attempt at adding contracts for first-class classes to Racket [Strickland and Felleisen 2010a], we imposed overheads of greater than 300% to some operations due to a hand-rolled proxying solution. This inefficiency sparked the creation of Racket's new system of proxies, dubbed chaperones and impersonators [Strickland et al. 2012], which is inspired by work in the Javascript community [Van Cutsem and Miller 2010; Austin et al. 2010]. Now, 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.

Local method calls have an additional overhead of 7% because local 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 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 varying numbers 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.

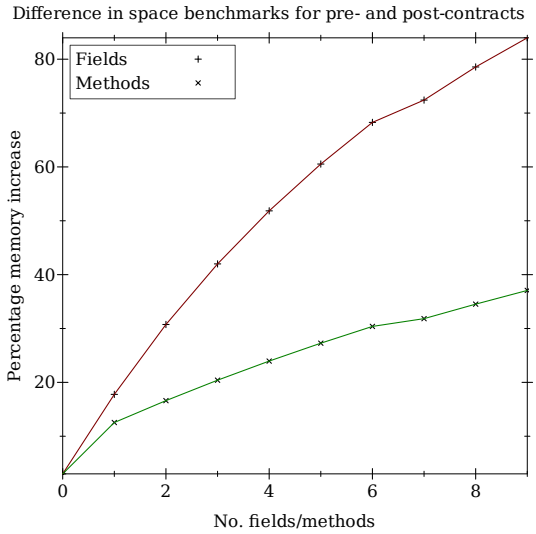


Fig. 28. Space benchmarks for classes

Figure 28 contains a graph of the resulting measurements. Each point represents the proportion of additional space required by adding contracts to classes. 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 contains methods. The large bump for the first added method represents parts of the class representation allocated when a given class contains methods.

On the other hand, the proportion of memory used by a class with contracts grows as more class features are added. This is expected, since the overhead of the contracts is outweighed by the other portions of the class representation when only a few methods or fields are present. Unfortunately, this also means that the addition of contracts for all methods can double the space used by a class with about 10 methods. For some applications, this may be unacceptable. We intend to investigate how to reduce the memory use of class contracts in future work.

8.2. 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[6] 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 did not use any dynamic checks to begin with.

Table II provides a breakdown of what kind of contracts we supply and the number of lines of code of both the original libraries and the changes we made. It ignores all contracts for non-OO features. The majority of these contracts protect classes and describe how a client should interact with instances and how a subclass should specialize behavior. Others apply to interfaces and mixins.

[6]A type-like contract uses type predicates such as *number?* to check arguments, results, and fields.

Table II. Classes, Interfaces, and Mixin Counts

Category	GUI	Framework	Total
Class	78	55	133
Interface	18	52	70
Mixin	0	47	47
All	96	154	250
Size	30,000 loc	20,000 loc	50,000 loc
Addition	2,700 loc	2,000 loc	4,700 loc
Subtraction	460 loc	n/a	460 loc

As seen in section 3, 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:

```
(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:

```
(define area⟨%⟩/c
  (class/c [get-parent (→m (or/c (is-a?/c area-container⟨%⟩) false/c))]
           [get-top-level-window (→m (or/c (is-a?/c frame%) (is-a?/c dialog%)))] ...))
```

```
(define window⟨%⟩/c
  (and/c area⟨%⟩/c
         (class/c [has-focus? (→m boolean?)]
                  [on-focus (→m any/c void?)]
                  (override [on-focus (→m any/c void?)]) ...)))
```

The contract for *window*⟨%⟩ 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.:

```
(define subwindow⟨%⟩/c (and/c subarea⟨%⟩/c window⟨%⟩/c))
```

Table III 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 III. Use of Contract Forms

	GUI	Framework	Total
method	1197	462	1659
init	390	0	390
absent	0	205	205
override	135	147	282
augment	0	59	59
inner	53	6	59
inherit	0	292	292
super	0	8	8
Size	30,000 loc	20,000 loc	50,000 loc
Addition	2,700 loc	2,000 loc	4,700 loc
Subtraction	460 loc	n/a	460 loc

Another insight concerns the use of specialization interfaces [Lamping 1993], especially in the GUI libraries. Recall from section 3 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.[7] 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.

[7] It also revealed the need for the **absent** clause, discussed in section 3.

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 test suite 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.

8.3. Performance Evaluation

To evaluate the performance of the contract system, we use the DrRacket [Findler et al. 2002] 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 IV. Contract performance (times in milliseconds)

run	max	mean	min	norm
No Checks or Contracts	747847	728874	722926	0.97
No Contracts	788179	749939	741486	1.00
Only GUI Contracts	809095	796642	792165	1.06
GUI and Framework Contracts	900266	821175	807671	1.09

Because the original GUI library uses dynamic checks in lieu of contracts, we also measure the amount of overhead introduced by these original checks. Table IV 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 29 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 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.

9. RELATED WORK

Parnas [1972] introduced the idea of describing expected behavior between different software components via specifications. Meyer [1992a] named these specifications

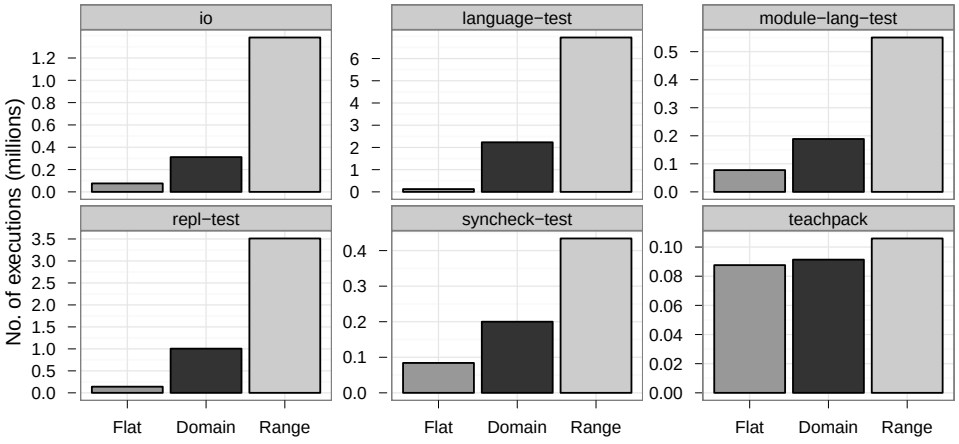


Fig. 29. Frequency of contract execution in test components

"contracts" and added them to his language Eiffel [Meyer 1992b]. He also coined the slogan "design by contract" to refer to the practice of writing contracts first.

Leavens et al. [2005] present JML, which goes beyond normal contract systems to include specification-only declarations and frame conditions. Leavens [2006] then shows how JML forces subtypes to inherit an appropriate combination of their supertypes' specifications in a way that enforces behavioral subtyping. Since behavioral subtyping is enforced, the information known about a value's static type can be used to provide modular reasoning about its behavior. As in Eiffel, however, specifications of higher-order behavior must be encoded by adding new interfaces to the type hierarchy.

Findler and Felleisen [2002] described a contract system for higher-order values, needed to protect higher-order functions and introduced the notions of contract boundaries and blame. Findler and Felleisen [2001] also discuss contract soundness for an object-oriented system, but their contract system, like Eiffel, provides only first-order checks on method entry and exit that are tightly coupled with the protected class.

Gray et al. [2005] use mirrors [Bracha and Ungar 2004] and contracts to enable interoperability between Java and Scheme by interposing dynamic checks on the boundaries between the two languages. Gray [2008] refines the idea of mirrors into guards and mimics to fix errors in blame allocation during cross-language inheritance in the previous work. Classes are not values that can flow between the languages in either system and the class hierarchy for a given program is static, but both systems provide behavior-based checking. Gray [2010] then uses guards and mimics to provide interoperability between Java and JavaScript. In this system, JavaScript objects defined at the top-level can be used as superclasses for Java classes and Java objects as prototypes for JavaScript objects. This means that the resulting class hierarchy is flexible but at the cost of expensive inter-language boundary crossings. In addition, all three pieces of work contain no efficiency or utility evaluation for the proposed systems.

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]. The idea to track negative information about labels is also inspired by research on type systems with extensible polymorphic records [Wand 1994; Cardelli and Mitchell 1991; Harper and Pierce 1991; Gaster and Jones 1996].

10. CONCLUSION

This paper presents a contract system for languages with first-class classes and provides both theoretic and practical validation. Our contract system can describe constraints on the behavior and state of objects and class values that cannot be checked immediately. To describe functions on classes, our contract system decouples contracts from classes. This separation allows the same contract to be shared between multiple classes or the same class to be protected with multiple different contracts. Most importantly, it empowers programmers to write contracts on class abstractions such as mixins. Finally, our contracts can express different restrictions on object interfaces and specialization interfaces.

Our contract system does not attempt to enforce behavioral subtyping in the class and interface hierarchies. While we consider this focus appropriate for a dynamically typed language, we plan to add facilities for the specification of behavioral subtyping, especially in the context of a typed language. In addition, the overhead incurred by our changes to the class system may be unacceptable for some uses. Developing alternative implementation strategies that reduce this overhead and make this contract design space-efficient is an open problem.

We validate our design by proving that our contracts assign correct blame for contract violations. Our proof of blame correctness is the first for a contract system for an object-oriented language. We validate our implementation by both measuring the runtime impact of our contracts and identifying the errors they reveal in our codebase by equipping a large code base with contracts. Last but not least, our use of class contracts pinpoints numerous inconsistencies between our code base and its documentation.

Acknowledgments The authors would like to thank Matthew Flatt (Utah) and Robby Findler (Northwestern) for discussions on first-class classes and class contracts. The anonymous TOPLAS reviewers provided valuable feedback concerning the presentation of this material.

REFERENCES

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

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

Davide Ancona, Giovanni Lagorio, and Elena Zucca. 2003. Jam—designing a Java extension with mixins. *Transactions on Programming Languages and Systems* 25, 5 (2003), 641–712.

Thomas H. Austin, Tim Disney, and Cormac Flanagan. 2010. *Virtual Values for Language Extension*. Technical Report UCSC-SOE-10-25. University of California Santa Cruz.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2004. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (LNCS)*, Vol. 3362. 49–69.

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

Gilad Bracha. 1992. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance*. Ph.D. Dissertation. University of Utah.

Gilad Bracha and William Cook. 1990. Mixin-Based Inheritance. In *Object-Oriented Programming, Systems, Languages, and Applications / European Conference on Object-Oriented Programming*. Ottawa, Canada, 303–311.

Gilad Bracha and David Ungar. 2004. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Object-Oriented Programming, Systems, Languages, and Applications*. 331–344.

Luca Cardelli and John C. Mitchell. 1991. Operations on records. *Mathematical Structures in Computer Science* 1 (1991), 3–48.

Manuela Carrillo-Castellón, Jesús García-Molina, Ernesto Pimentel, and Israel Repiso. 1996. Design by contract in Smalltalk. *Journal of Object-Oriented Programming* 7, 9 (1996), 23–28.

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

Christos Dimoulas, Robert B. Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *Symposium on Principles of Programming Languages*. 215–226.

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

Erik Ernst. 1999. *gbeta—a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph.D. Dissertation. University of Aarhus, Århus, Denmark.

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17, 1-3 (1991), 35–75.

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

Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *Functional and Logic Programming (LNCS)*, Vol. 3945. 226–241.

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

Robert Bruce Findler and Matthias Felleisen. 2001. Contract soundness for object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications*. 1–15.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *International Conference on Functional Programming*. 48–59.

Kathleen Fisher and John Reppy. 2004. A typed calculus of traits. In *Workshop on Foundations of Object-Oriented Languages*.

David Flanagan and Yukihiro Matsumoto. 2008. *The Ruby Programming Language*. O'Reilly.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. 2006. Scheme with Classes, Mixins, and Traits. In *Asian Symposium on Programming Languages and Systems (LNCS)*, Vol. 4279. 270–289.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *Symposium on Principles of Programming Languages*. 171–183.

Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Inc. http://racket-lang.org/tr1/.

Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants*. Technical Report NOTTCS-TR-96-3. University of Nottingham.

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

David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. 2004. Super and Inner—Together at Last!. In *Object-Oriented Programming, Systems, Languages, and Applications*. 116–129.

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

Kathryn E. Gray. 2008. Safe Cross-Language Inheritance. In *European Conference on Object-Oriented Programming (LNCS)*, Vol. 5142. 52–75.

Kathryn E. Gray. 2010. Interoperability in a Scripted World: Putting Inheritance and Prototypes Together. In *Foundations of Object-Oriented Languages*.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-Grained Interoperability through Contracts and Mirrors. In *Object-Oriented Programming, Systems, Languages, and Applications*. 231–245.

Robert Harper and Benjamin Pierce. 1991. A Records Calculus Based on Symmetric Concatenation. In *Symposium on Principles of Programming Languages*. 131–142.

Murat Karaorman, Urs Hölzle, and John Bruno. 1999. jContractor: A Reflective Java Library to Support Design By Contract. In *Proceedings of Meta-Level Architectures and Reflection (LNCS)*, Vol. 1616. 175–196.

Michael Kölling and John Rosenberg. 1997. *Blue: Language Specification, version 0.94*. http://www.sd.monash.edu.au/blue/

Reto Kramer. 1998. iContract: The Java Design by Contract Tool. In *Technology of Object-Oriented Languages and Systems*. 295.

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

Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. 1977. Report on the programming language Euclid. *SIGPLAN Not.* 12, 2 (Feb. 1977), 1–79.

Gary T. Leavens. 2006. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *International Conference on Formal Engineering Methods (LNCS)*, Vol. 4260. 2–34.

Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming* 55, 1-3 (2005), 185–208.

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

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

David C. Luckham and Friedrich W. von Henke. 1985. An overview of Anna, a specification language for Ada. *IEEE Software* 2 (March 1985), 9–23.

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

Bertrand Meyer. 1992a. Applying Design by Contract. *IEEE Computer* 25, 10 (1992), 40–51.

Bertrand Meyer. 1992b. *Eiffel: The Language*. Prentice Hall.

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

David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15 (1972), 1053–1058. Issue 12.

Reinhold Plösch. 1997. Design by Contract for Python. In *IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference*. 213–219.

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science* (1977), 223–255.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. 2003. Traits: Composable Units of Behavior. In *European Conference on Object-Oriented Programming (LNCS)*, Vol. 2743. 248–274.

Marco Servetto and Elena Zucca. 2010. MetaFJig: a meta-circular composition language for Java-like classes. In *Object-Oriented Programming, Systems, Languages, and Applications*. 464–483.

T. Stephen Strickland and Matthias Felleisen. 2010a. Contracts for First-Class Classes. In *Dynamic Languages Symposium*. 97–111.

T. Stephen Strickland and Matthias Felleisen. 2010b. Nested and Dynamic Contract Boundaries. In *IFL 2009 (LNCS)*, Vol. 6041. 141–158.

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

Clemens Szyperski. 1997. *Component Software*. Addison-Wesley.

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

Mitchell Wand. 1994. Type Inference for Objects with Instance Variables and Inheritance. In *Theoretical Aspects of Object-Oriented Programming*. 97–120.

A. META-FUNCTIONS FOR CFCC

A.1. The $\mathcal{CV}$ Meta-Function

$$
\begin{array}{lll}
\mathcal{CV}[\![\textbf{object\%}]\!] & = & \mathcal{T}rue \\
\mathcal{CV}[\![\gamma]\!] & = & \mathcal{CV}[\![\gamma']\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{CV}[\![\gamma]\!] & = & \mathcal{CV}[\![\gamma']\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override} \\
 & & \qquad\qquad\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
\mathcal{CV}[\![v]\!] & = & \mathcal{F}alse \quad \text{in any other case}
\end{array}
$$

A.2. The $\mathcal{M}ethods$ Meta-Function

$$
\begin{array}{lll}
\mathcal{M}ethods[\![\gamma]\!] & = & \{m_1 \ldots\} \cup \mathcal{M}ethods[\![\gamma']\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{M}ethods[\![\textbf{object}(\gamma)]\!] & = & \mathcal{M}ethods[\![\gamma]\!] \\
\mathcal{M}ethods[\![\gamma]\!] & = & \mathcal{M}ethods[\![\gamma']\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\ [m_p\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad\qquad \textbf{override} \\
 & & \qquad\qquad\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
\mathcal{M}ethods[\![v]\!] & = & \varnothing \\
 & & \text{in any other case}
\end{array}
$$

A.3. The Class Identifier Substitution Meta-Function

$$
\begin{array}{lll}
\{\iota'/^{id}\iota\}e & = & e \\
 & & \text{where } e = \textbf{class}\{\ e' \\
 & & \qquad\qquad \textbf{inherit}\ m_{i_1} \ldots \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\ [m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ e_{p_1}] \ldots \\
 & & \qquad\qquad \textbf{override} \\
 & & \qquad\qquad\ [m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ e_{o_1}] \ldots\} \\
\{\iota'/^{id}\iota\}e & = & e \\
 & & \text{where } e = \textbf{class/v}^{\iota}\{\ \gamma \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\{\iota'/^{id}\iota\}\textbf{isend}^{\iota}(e_o, m, e_1 \ldots) & = & \textbf{isend}^{\iota'}(\{\iota'/^{id}\iota\}e_o, m, \{\iota'/^{id}\iota\}e_1 \ldots) \\
\{\iota'/^{id}\iota\}\textbf{super}^{\iota}(e_o, m, e_1 \ldots) & = & \textbf{super}^{\iota'}(\{\iota'/^{id}\iota\}e_o, m, \{\iota'/^{id}\iota\}e_1 \ldots)
\end{array}
$$

for the remaining elements of e and κ the meta-function is a homomorphism

A.4. The $\mathcal{O}bject$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{O}bject[\![\textbf{object}(\gamma)]\!] & = & \mathcal{T}rue \\
 & & \text{if} \mathcal{C}V[\![\gamma]\!] \\
\mathcal{O}bject[\![\gamma]\!] & = & \mathcal{O}bject[\![\gamma']\!] \\
 & & \text{where}\, \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\quad [m_{p_1}\ (\kappa^{p_1}_{this}\ \kappa^{p_1}_1 \ldots \mapsto \kappa^{p_1}_r)^{k,l}_j] \ldots \\
 & & \qquad\qquad \textbf{override} \\
 & & \qquad\qquad\quad [m_{o_1}\ (\kappa^{o_1}_{this}\ \kappa^{o_1}_1 \ldots \mapsto \kappa^{o_1}_r)^{l,k}_j] \ldots\} \\
\mathcal{O}bject[\![v]\!] & = & \mathcal{F}alse \quad \text{in any other case}
\end{array}
$$

A.5. The $\mathcal{I}s$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{I}s[\![\gamma, \iota]\!] & = & \mathcal{T}rue \\
 & & \text{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\quad [m_1(this_1\ x^1_1 \ldots)\ e_1] \ldots\} \\
\mathcal{I}s[\![\gamma, \iota]\!] & = & \mathcal{I}s[\![\gamma', \iota]\!] \\
 & & \text{where}\, \gamma = \textbf{class/v}^{\iota'}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\quad [m_1(this_1\ x^1_1 \ldots)\ e_1] \ldots\} \\
 & & \text{and}\, \iota \neq \iota' \\
\mathcal{I}s[\![\textbf{object}(\gamma), \iota]\!] & = & \mathcal{I}s[\![\gamma, \iota]\!] \\
\mathcal{I}s[\![\gamma, \iota]\!] & = & \mathcal{I}s[\![\gamma', \iota]\!] \\
 & & \text{where}\, \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\quad [m_{p_1}\ (\kappa^{p_1}_{this}\ \kappa^{p_1}_1 \ldots \mapsto \kappa^{p_1}_r)^{k,l}_j] \ldots \\
 & & \qquad\qquad \textbf{override} \\
 & & \qquad\qquad\quad [m_{o_1}\ (\kappa^{o_1}_{this}\ \kappa^{o_1}_1 \ldots \mapsto \kappa^{o_1}_r)^{l,k}_j] \ldots\} \\
\mathcal{I}s[\![\textbf{object\%}, \iota]\!] & = & \mathcal{F}alse
\end{array}
$$

A.6. The $\mathcal{G}etS$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{G}etS[\![\gamma, \iota]\!] & = & \gamma' \\
 & & \text{where}\, \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\quad [m_1(this_1\ x^1_1 \ldots)\ e_1] \ldots\} \\
\mathcal{G}etS[\![\gamma, \iota]\!] & = & \mathcal{G}etS[\![\gamma', \iota]\!] \\
 & & \text{where}\, \gamma = \textbf{class/v}^{\iota'}\{\ \gamma' \\
 & & \qquad\qquad \textbf{methods} \\
 & & \qquad\qquad\quad [m_1(this_1\ x^1_1 \ldots)\ e_1] \ldots\} \\
 & & \text{and}\, \iota \neq \iota' \\
\mathcal{G}etS[\![\textbf{object}(\gamma), \iota]\!] & = & \mathcal{G}etS[\![\gamma, \iota]\!] \\
\mathcal{G}etS[\![\gamma, \iota]\!] & = & \mathcal{G}etS[\![\gamma', \iota]\!] \\
 & & \text{where}\, \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad\qquad \textbf{public} \\
 & & \qquad\qquad\quad [m_{p_1}\ (\kappa^{p_1}_{this}\ \kappa^{p_1}_1 \ldots \mapsto \kappa^{p_1}_r)^{k,l}_j] \ldots \\
 & & \qquad\qquad \textbf{override} \\
 & & \qquad\qquad\quad [m_{o_1}\ (\kappa^{o_1}_{this}\ \kappa^{o_1}_1 \ldots \mapsto \kappa^{o_1}_r)^{l,k}_j] \ldots\}
\end{array}
$$

A.7. The $\mathcal{G}et$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{G}et[\![\gamma, \iota]\!] & = & \gamma \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \gamma' \\
 & & \qquad \textbf{methods} \\
 & & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{G}et[\![\gamma, \iota]\!] & = & \mathcal{G}et[\![\gamma', \iota]\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \gamma' \\
 & & \qquad \textbf{methods} \\
 & & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \text{and } \iota \neq \iota' \\
\mathcal{G}et[\![\textbf{object}(\gamma), \iota]\!] & = & \mathcal{G}et[\![\gamma, \iota]\!] \\
\mathcal{G}et[\![\gamma, \iota]\!] & = & \mathcal{G}et[\![\gamma', \iota]\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \gamma' \\
 & & \qquad \textbf{public} \\
 & & \qquad\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

B. MINOR META-FUNCTIONS FOR CFCC WITH ANNOTATIONS

B.1. The $\mathcal{C}V$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{C}V[\![\|\textbf{object\%}\|^{\vec{l}}]\!] & = & \mathcal{T}rue \\
\mathcal{C}V[\![\|\gamma\|^{\vec{k}}]\!] & = & \mathcal{C}V[\![\|\gamma'\|^{\vec{l}}]\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{C}V[\![\|\gamma\|^{\vec{k}}]\!] & = & \mathcal{C}V[\![\|\gamma'\|^{\vec{l}}]\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public} \\
 & & \qquad\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
\mathcal{C}V[\![v]\!] & = & \mathcal{F}alse \\
 & & \text{in any other case}
\end{array}
$$

B.2. The $\mathcal{M}ethods$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{M}ethods[\![\|\gamma\|^{\vec{k}}]\!] & = & \{m_1 \ldots\} \cup \mathcal{M}ethods[\![\|\gamma'\|^{\vec{l}}]\!] \\
 & & \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{M}ethods[\![\|\textbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}]\!] & = & \mathcal{M}ethods[\![\|\gamma\|^{\vec{l}}]\!] \\
\mathcal{M}ethods[\![\|\gamma\|^{\vec{k}}]\!] & = & \mathcal{M}ethods[\![\|\gamma'\|^{\vec{l}}]\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public} \\
 & & \qquad\ [m_p\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
\mathcal{M}ethods[\![v]\!] & = & \varnothing \quad \text{in any other case}
\end{array}
$$

B.3. The Class Identifier Substitution Meta-Function

$$
\begin{array}{lcl}
\{\iota'/^{id}\iota\}e & = & e \\
 & & \text{where } e = \textbf{class}\{\ e' \\
 & & \qquad \textbf{inherit}\ m_{i_1} \ldots \\
 & & \qquad \textbf{public} \\
 & & \qquad\ [m_{p_1}(this_{p_1}\ x_1^{p_1} \ldots)\ e_{p_1}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\ [m_{o_1}(this_{o_1}\ x_1^{o_1} \ldots)\ e_{o_1}] \ldots\} \\
\{\iota'/^{id}\iota\}e & = & e \\
 & & \text{where } e = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\ [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\{\iota'/^{id}\iota\}\textbf{isend}^{\iota}(e_o, m, e_1 \ldots) & = & \textbf{isend}^{\iota'}(\{\iota'/^{id}\iota\}e_o, m, \{\iota'/^{id}\iota\}e_1 \ldots) \\
\{\iota'/^{id}\iota\}\textbf{super}^{\iota}(e_o, m, e_1 \ldots) & = & \textbf{super}^{\iota'}(\{\iota'/^{id}\iota\}e_o, m, \{\iota'/^{id}\iota\}e_1 \ldots)
\end{array}
$$

for the remaining elements of e and κ the meta-function is a homomorphism

B.4. The $\mathcal{O}bject$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{O}bject[\![\|\textbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}]\!] & = & \mathcal{T}rue \\
 & & \text{if } \mathcal{C}V[\![\|\gamma\|^{\vec{l}}]\!] \\
\mathcal{O}bject[\![\|\gamma\|^{\vec{k}}]\!] & = & \mathcal{O}bject[\![\|\gamma'\|^{\vec{l}}]\!] \\
 & & \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public} \\
 & & \qquad\ [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\ [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
\mathcal{O}bject[\![v]\!] & = & \mathcal{F}alse \quad \text{in any other case}
\end{array}
$$

B.5. The $\mathcal{I}s$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{I}s[\![\|\gamma\|^{\vec{k}}, \iota]\!] & = & \mathcal{T}rue \\
 & & \quad \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\quad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{I}s[\![\|\gamma\|^{\vec{k}}, \iota]\!] & = & \mathcal{I}s[\![\|\gamma'\|^{\vec{l}}, \iota]\!] \\
 & & \quad \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\quad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \quad \text{and } \iota \neq \iota' \\
\mathcal{I}s[\![\|\textbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}, \iota]\!] & = & \mathcal{I}s[\![\|\gamma\|^{\vec{l}}, \iota]\!] \\
\mathcal{I}s[\![\|\gamma\|^{\vec{k}}, \iota]\!] & = & \mathcal{I}s[\![\|\gamma'\|^{\vec{l}}, \iota]\!] \\
 & & \quad \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public} \\
 & & \qquad\quad [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\quad [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\} \\
\mathcal{I}s[\![\|\textbf{object\%}\|^{\vec{l}}, \iota]\!] & = & \mathcal{F}alse
\end{array}
$$

B.6. The $\mathcal{G}etS$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{G}etS[\![\|\gamma\|^{\vec{k}}, \iota, l]\!] & = & \|\gamma'\|^{l \oplus \vec{l}} \\
 & & \quad \text{where } \gamma = \textbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\quad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{G}etS[\![\|\gamma\|^{\vec{k}}, \iota, l]\!] & = & \mathcal{G}etS[\![\|\gamma'\|^{\vec{l}}, \iota, l \otimes \vec{k}\ ]\!] \\
 & & \quad \text{where } \gamma = \textbf{class/v}^{\iota'}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{methods} \\
 & & \qquad\quad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \quad \text{and } \iota \neq \iota' \\
\mathcal{G}etS[\![\|\textbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}, \iota, l]\!] & = & \mathcal{G}etS[\![\|\gamma\|^{\vec{l}}, \iota, l \otimes \vec{k}\ ]\!] \\
\mathcal{G}etS[\![\|\gamma\|^{\vec{k}}, \iota, l]\!] & = & \mathcal{G}etS[\![\|\gamma'\|^{\vec{l}}, \iota, l \otimes \vec{k}\ ]\!] \\
 & & \quad \text{where } \gamma = \textbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \textbf{public} \\
 & & \qquad\quad [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \textbf{override} \\
 & & \qquad\quad [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$

B.7. The $\mathcal{G}et$ Meta-Function

$$
\begin{array}{lcl}
\mathcal{G}et[\![\|\gamma\|^{\vec{k}}, \iota, l]\!] & = & \|\gamma\|^{l \oplus \vec{k}} \\
 & & \text{where } \gamma = \mathbf{class/v}^{\iota}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \mathbf{methods} \\
 & & \qquad\quad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
\mathcal{G}et[\![\|\gamma\|^{\vec{k}}, \iota, l]\!] & = & \mathcal{G}et[\![\|\gamma'\|^{\vec{l}}, \iota, l \otimes \vec{k}\ ]\!] \\
 & & \text{where } \gamma = \mathbf{class/v}^{\iota'}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \mathbf{methods} \\
 & & \qquad\quad [m_1(this_1\ x_1^1 \ldots)\ e_1] \ldots\} \\
 & & \text{and } \iota \neq \iota' \\
\mathcal{G}et[\![\|\mathbf{object}(\|\gamma\|^{\vec{l}})\|^{\vec{k}}, \iota, l]\!] & = & \mathcal{G}et[\![\|\gamma\|^{\vec{l}}, \iota, l \otimes \vec{k}\ ]\!] \\
\mathcal{G}et[\![\|\gamma\|^{\vec{k}}, \iota, l]\!] & = & \mathcal{G}et[\![\|\gamma'\|^{\vec{l}}, \iota, l \otimes \vec{k}\ ]\!] \\
 & & \text{where } \gamma = \mathbf{G}\{\ \|\gamma'\|^{\vec{l}} \\
 & & \qquad \mathbf{public} \\
 & & \qquad\quad [m_{p_1}\ (\kappa_{this}^{p_1}\ \kappa_1^{p_1} \ldots \mapsto \kappa_r^{p_1})_j^{k,l}] \ldots \\
 & & \qquad \mathbf{override} \\
 & & \qquad\quad [m_{o_1}\ (\kappa_{this}^{o_1}\ \kappa_1^{o_1} \ldots \mapsto \kappa_r^{o_1})_j^{l,k}] \ldots\}
\end{array}
$$