RICE UNIVERSITY

# Behavioral Software Contracts

by

**Robert Bruce Findler**

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:

Matthias Felleisen
Professor of Computer Science

Robert "Corky" Cartwright
Professor of Computer Science

Keith D. Cooper
Professor of Computer Science

Michael Barlow
Assistant Professor of Linguistics

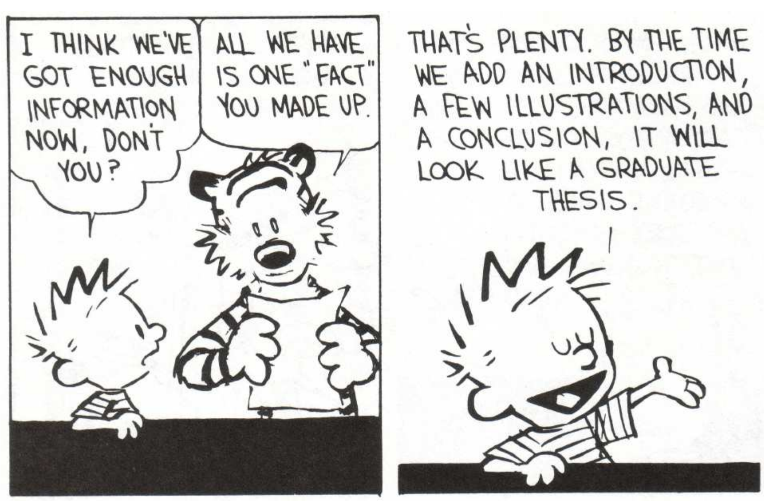
HOUSTON, TEXAS

April, 2002

# Behavioral Software Contracts

by

**Robert Bruce Findler**



# Behavioral Software Contracts

by

**Robert Bruce Findler**


## Abstract

To sustain a market for software components, component producers and consumers must agree on contracts. These contracts must specify each party's obligations. To ensure that both sides meet their obligations, they must also agree on standards for monitoring contracts and assigning blame for contract violations.

This dissertation explores these issues for contracts that specify the sequential behavior of methods and procedures as pre- and post-conditions. In the process, it makes three main contributions:

- First, this dissertation shows how existing contract checking systems for object-oriented languages incorrectly enforce contracts in the presence of subtyping. This dissertation shows how to check such contracts properly.

- Second, this dissertation shows how to enforce pre- and post-condition style contracts on higher-order procedures and correctly assign blame for contract violations in that context.

- Finally, this dissertation lays the groundwork for a theory of contract checking, in the spirit of the theory for type checking. In particular, it states and proves the first soundness result for contracts, guaranteeing that the contract checker properly enforces contracts and properly assigns blame for contract violations.

# Acknowledgments

I owe my entire academic life to my advisor, Matthias Felleisen. His tireless struggle to train me has made an incredible difference in my life (and I certainly did not make it easy for him!). Thank you, Matthias!

During my years at Rice, I worked especially closely with Matthew Flatt and Shriram Krishnamurthi. Their patient guidance and support was invaluable in my development, both as a programmer and as a researcher.

Thanks also to the others I worked closely with: Kevin Charter, John Clements, Cormac Flanagan, Paul Graunke, Philippe Meunier, Jamie Raymond and Paul Steckler.

Mario Latendresse helped me uncover the flaw in object-oriented contract checkers. Clemens Szyperski read many early versions of chapters in this dissertation and gave me many excellent ideas along the way. Daniel Jackson also had great comments on an early drafts of this work. Gregory T. Sullivan pointed out a flaw in my formulation of contract soundness. Thanks, all.

A special thanks to Ian Barland who was always available to have some coffee, making many an otherwise horrible day manageable. Thanks to Dorai Sitaram for teaching me the wonders of vegetarianism and helping a great deal with the introduction to this dissertation. Darnell, Iva Jean, and Rhonda were always ready with the perfect word of encouragement, humor, or guidance, and I will always be grateful.

My mother and father and my brothers have always been a source of great love and encouragement. Their love shaped me into the person I am today.

Finally, I want to thank my wife, Hsing-Huei Huang, for her love, support, and encouragement. She makes my life worthwhile. This is for you, baby!

# Contents

Abstract iii

Acknowledgments iv

Table of Contents v

List of Illustrations viii

**1 Introduction 1**

1.1 Components and DrScheme . . . . . 3

1.2 Class and Function Contracts from DrScheme . . . . . 4

1.2.1 Flat Contracts . . . . . 4

1.2.2 Class-based Contracts . . . . . 7

1.2.3 Higher-order Contracts . . . . . 10

1.3 Thesis Statement and Roadmap . . . . . 11

**2 Behavioral Subtyping and Related Work 13**

2.1 Behavioral Contracts . . . . . 13

2.2 Behavioral Subtyping . . . . . 14

2.3 Contracts and Behavioral Subtypes . . . . . 15

2.4 Problems with Prior Work . . . . . 18

2.4.1 Jass . . . . . 22

2.5 Properly Monitoring Contracts . . . . . 22

**3 Contract Compilation 25**

3.1 How to Check Contracts and Assign Blame . . . . . 25

3.1.1 Flat Contract Checking . . . . . 26

3.1.2 Interface Implementation . . . . . . . . . . . . . . . . . . . . . . . . 27
3.1.3 Class Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.1.4 Multiple Inheritance . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.2 Environmental Considerations . . . . . . . . . . . . . . . . . . . . . . . 32
3.3 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

**4 Contract Soundness 34**

4.1 From Type Soundness to Contract Soundness . . . . . . . . . . . . . . . . 34
4.2 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Type Elaboration . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.4 Contract Elaboration . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.6 Contract Soundness . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

**5 Contracts for Higher-Order Functions 67**

5.1 From First-Order Function Contracts
to Higher-Order Function Contracts . . . . . . . . . . . . . . . . . . . . 68
5.2 Example Contracts . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.2.1 Contracts: A First Look . . . . . . . . . . . . . . . . . . . . . . . 70
5.2.2 Enforcement at First-Order Types . . . . . . . . . . . . . . . . . . . 72
5.2.3 Blame and Contravariance . . . . . . . . . . . . . . . . . . . . . . 76
5.2.4 First-class Contracts . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.2.5 Callbacks and Stateful Contracts . . . . . . . . . . . . . . . . . . . 80
5.3 Contract Calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4 Contract Monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.5 Contract Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.6 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.7 Dependent Contracts . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.8 Tail Recursion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

5.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

**6 Conclusions and Future Work** **105**

**Bibliography** **107**

# Illustrations


1.1 GUI Test Suite Contract . . . . . . . . . . 5

1.2 Relative Percentages Contract . . . . . . . . . . 6

1.3 Editor Mixin Contract . . . . . . . . . . 7

1.4 Editor Mixin Extensions . . . . . . . . . . 9

1.5 Mixin Composition . . . . . . . . . . 10

1.6 Controlling DrScheme's Printer . . . . . . . . . . 11

2.1 The Behavioral Subtyping Condition . . . . . . . . . . 15

2.2 The Behavioral Subtyping Condition, Generalized to Multiple Inheritance . 15

2.3 Behavioral Subtyping in Interfaces . . . . . . . . . . 18

2.4 Delayed, Incorrect Explanation for Contract Violation . . . . . . . . . . 21

2.5 Hierarchy Checking . . . . . . . . . . 23

3.1 Section 3.1 Overview . . . . . . . . . . 26

3.2 Pre- and Post-condition Checking . . . . . . . . . . 27

3.3 Hierarchy Checking . . . . . . . . . . 29

3.4 Hierarchy Checking for Multiple Inheritance . . . . . . . . . . 31

4.1 Contract Java syntax; before and after contracts are compiled away . . . . . 36

4.2 Predicates and relations in the model of Contract Java , i . . . . . . . . . . 37

4.3 Predicates and relations in the model of Contract Java, ii . . . . . . . . . . 38

4.4 Context-sensitive Checks and Type Elaboration Rules for Contract Java, i . 42

4.5 Context-sensitive Checks and Type Elaboration Rules for Contract Java, ii . 43
4.6 Blame Compilation, i . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.7 Blame Compilation, ii . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.8 Blame Compilation, iii . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.9 Blame Compilation, iv . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.10 Example Hierarchy Diagram . . . . . . . . . . . . . . . . . . . . . . . . 54
4.11 Elaborated Console Example . . . . . . . . . . . . . . . . . . . . . . . . 58
4.12 Operational semantics for Contract Java . . . . . . . . . . . . . . . . . . 59

5.1 Contract specified with *add-panel* . . . . . . . . . . . . . . . . . . . . . 72
5.2 Contract manually distributed . . . . . . . . . . . . . . . . . . . . . . . . 74
5.3 Abstraction for Predicate Contracts . . . . . . . . . . . . . . . . . . . . . 78
5.4 Preferences panel contract, protecting the panel . . . . . . . . . . . . . . 79
5.5 $\lambda^{\text{CON}}$ Syntax and Types . . . . . . . . . . . . . . . . . . . . . . . . 82
5.6 $\lambda^{\text{CON}}$ Evaluation Contexts and Values . . . . . . . . . . . . . . . . 83
5.7 Reduction Semantics of $\lambda^{\text{CON}}$ . . . . . . . . . . . . . . . . . . . . . 84
5.8 $\lambda^{\text{CON}}$ Type Rules . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.9 Obligation Expression Insertion . . . . . . . . . . . . . . . . . . . . . . 87
5.10 Monitoring Contracts in $\lambda^{\text{CON}}$ . . . . . . . . . . . . . . . . . . . . 90
5.11 Reducing *sqrt* in $\lambda^{\text{CON}}$ . . . . . . . . . . . . . . . . . . . . . . . . 91
5.12 Reducing *sqrt* with *wrap* . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.13 Contract Compiler Wrapping Function . . . . . . . . . . . . . . . . . . . 93
5.14 Contract Compiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.15 Simulation between $\mathcal{E}_{fw}$ and $\mathcal{E}_{fw}$ . . . . . . . . . . . . . . . . . . . . . 95
5.16 Evaluators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.17 Dependent Function Contracts for $\lambda^{\text{CON}}$ . . . . . . . . . . . . . . . . . . 102

# Chapter 1

# Introduction

Modern software development often requires collaboration between independently operating groups of developers. These developers publish components and extensions that others combine to form a working system. McIlroy [39] first proposed the idea of software components in 1969. In a marketplace with reusable components, software manufacturers would produce software components with well-specified interfaces. Developers would assemble systems from these off-the-shelf components, possibly adapting some with wrapper code, instantiating abstractions in some, and even adding a few new ones. If a component were faulty, a developer could replace it with a different one. If a manufacturer were to improve a component, a developer could improve the final product by replacing the link to the old component with a link to the new one.

To make such a component marketplace work, components must come with interfaces that specify their key properties. These interfaces record contracts between the component producer and component consumer. Beugnard et al [5] list four levels of component contracts:

- syntactic contracts, *e.g.*, type signatures,
- behavioral contracts, *e.g.*, pre- and post-condition invariants, which state semantic properties that augment the languages type specifications,
- sequencing contracts, *e.g.*, thread synchronization and sequencing constraints, and
- quality of service contracts, *e.g.*, time and space guarantees.

In addition to recording the contracts between components, developers must agree on a mechanism for enforcing the contracts and assigning blame* for contract violations. For example, if the pre-condition of a behavioral contract fails, the developer who wrote the call to the corresponding method or procedure is blamed. If a post-condition fails, the developer who wrote the method or procedure itself is blamed. Properly assigned blame enables developers to quickly ascertain which component is faulty and then either fix the problem or replace the faulty component.

Run-time enforcement of behavioral contracts has been widely studied [1, 36, 40, 41, 44] and has a standard interpretation. Each method or procedure is annotated with a pre-condition and a post-condition. These conditions are effect-free program fragments that are evaluated when a method or procedure is called and when it returns. A pre-condition is evaluated when a method or procedure is called and if it produces `false`, the caller failed to establish the required conditions. Symmetrically, a post-condition is evaluated when a method or procedure returns and it indicates if the method or procedure itself was successful.

The remainder of the introduction is divided into two parts. The first part presents an overview of the component structure of DrScheme and explains how the component organization has facilitated program development. The second part consists of a series of contract examples drawn from DrScheme that motivate the remainder of the dissertation.

**Note.** In principle, one could try to prove the correctness of behavioral contracts. For example, the Extended Static Checking group has developed verification tools for Java and Modula 3 [8] that attempt to prove that certain behavioral properties are always satisfied. In general, however, the languages used to express behavioral contracts are so rich that it is intractably difficult to verify statically that the contracts of a component are never violated. In fact, the Extended Static Checking group's tools are neither complete nor sound, that is,

*I believe that a certain amount of accountability and pride in quality craftsmanship is critical to good software production. Thus, when I use the term "blame" I mean that the programmer should be held accountable for shoddy craftsmanship.

the tools may indicate that errors exist in correct code, or may indicate that no errors exist in incorrect code. Furthermore, most tools that do attempt to prove behavioral contracts correct are computationally expensive. Finally, these tools require training in logic and program verification that most programmers do not possess.

## 1.1 Components and DrScheme

Our source of examples for component-oriented programming is DrScheme [12], a programming environment for Scheme. It supports two forms of extension, TeachPacks and tools, each of which plays the role of a component deployment context.

**TeachPacks** DrScheme initially provides a language targeted at beginners. This language is small and designed for pedagogic clarity at the expense of expressiveness. As students learn more about programming, DrScheme's language adapts to the student by providing the students with more constructs. By allowing the student to progress along a tower of languages, DrScheme shields the student from the complexity of a professional's programming environment and still allows the student enough computational power to learn the fundamental principles of computation.

One cost of this simplicity is that instructors cannot assign exercises that involve graphics, networking, or other advanced features of the programming language to motivate students. To address this problem, DrScheme provides TeachPacks. A TeachPack is a series of definitions written in the full programming language that are dynamically injected into the students' programming language as new primitives. Typically, TeachPacks define a few exercise-specific GUI or networking primitives that hide the full complexity of those libraries, and yet still allow the student to use these features to solve problems.

**Tools** DrScheme's tools interface allows developers to enhance the core programming environment with new teaching languages, program analyses, syntax checkers, algebraic steppers, and other extensions. These extensions are loaded as DrScheme starts up, but the exact set of tools is not written into DrScheme's source code. Instead, as DrScheme starts up, it discovers which tools have been installed and loads them. This allows third party developers to provide tools independently of the main DrScheme releases.

To support TeachPacks and tools, DrScheme requires a powerful form of program extension. TeachPacks are dynamically linked multiple times, and tools are dynamically linked with externally specified imports. To support both TeachPacks and tools, DrScheme uses units [15], a software component mechanism designed as part of the MzScheme [14] programming language.

## 1.2 Class and Function Contracts from DrScheme

This section motivates the remainder of this dissertation with example contracts from DrScheme. The contracts range from simple predicates on flat values to the invariants of certain object-oriented design patterns [18] to restrictions on the arguments and results of higher-order functions. Most of the contracts presented here occur at the interface between DrScheme and the extensions described in the previous section.

### 1.2.1 Flat Contracts

For a first example, we turn to DrScheme's automatic GUI test suite library. It contains functions that simulate user actions like button clicks and menu selections. These operations have contracts that guarantee that the GUI's state is amenable to the test action.

The contract for *test:button-press*, the operation that simulates a button click, is shown in figure 1.1. It accepts either a string or a button as an argument. In both cases, the front-most window must be a DrScheme window (the *get-top-level-focus-window* primitive returns #f if the front-most window is not a DrScheme window). If the argument is a string, the front-most window must contain a button whose label is the string. If the argument is a button, the front-most window must contain that button. In both cases, the button

```
;; button-press-argument-okay? : (union string button) → boolean
;; determines if the argument to test:button-press satisfies its contract
(define (button-press-argument-okay? arg)
  (let ([top-level-focus-window (get-top-level-focus-window)])
    (cond
      [(string? arg)
       (and top-level-focus-window
            (frame-has? top-level-focus-window
                        (λ (x)
                          (and (is-a? x button%)
                               (string=? (send x get-label) arg)
                               (send x is-enabled?)
                               (send x is-shown?)))))]
      [(is-a? arg button%)
       (and top-level-focus-window
            (frame-has? top-level-focus-window
                        (λ (x)
                          (and (eq? x arg)
                               (send x is-enabled?)
                               (send x is-shown?)))))])))

;; frame-has? : (instanceof frame%) (area<%> → boolean) → boolean
;; determines if the frame contains a child that satisfies p.
(define (frame-has? frame p)
  (let test ([i frame])
    (or (p i)
        (and (is-a? i area-container<%>)
             (ormap test (send i get-children))))))
```

Figure 1.1: GUI Test Suite Contract

must be both visible and enabled. The *button-press-argument-okay?* procedure returns a boolean indicating the fitness of that value as an argument to *test:button-press*.

Formulating this contract in most type systems is not possible and proving it with a theorem prover is nearly intractable, because it depends on the state of the GUI which, in turn, depends on the sequence of user inputs submitted to DrScheme. In fact, when the test suite is running, it is possible to get spurious test failures if the person running the test suite interrupts the simulated stream of events with real mouse clicks. This causes DrScheme to

```
;; set-percentages : (listof number) → void
(define (set-percentages ps)
  (unless (and (list? ps)
               (andmap number? ps)
               (= (length ps) (length (send this get-children)))
               (= 1 (apply + ps))
               (andmap positive? ps))
    (error 'set-percentages
           "expected list of positive numbers that sum to 1, got: ~e"
           ps))
  (set! percentages (map make-percentage ps))
  (send this reflow-container))
```

Figure 1.2: Relative Percentages Contract

enter an unexpected state where the contracts do not hold.

Such failures are a consequence of the test suite architecture. After a simulated event is put into the event queue, DrScheme cannot distinguish it from an actual event. This property is important for the integrity of the test suite, but does have the negative consequence that, while the test suite is running, the programmer must not manipulate DrScheme's GUI.

DrScheme also contains contracts that are independent of user input, but proving them correct goes beyond the capabilities of traditional type-systems. For a second example, consider the function in figure 1.2. This function sets the relative percentages of the draggable windows in the main DrScheme frame. The boxed portion of the function checks the contract and the unboxed portion performs the real work of the function. The function's argument is a list of numbers, one for each subwindow. Since each subwindow's percentage must be specified, the list's length must match the number of subwindows. Additionally, since each number is treated as a percentage of the size of the total window, each percentage must be a positive number and together they must sum to 1. Most practical type systems are currently unable to express the fact that the length of two lists must match, let alone that a series of numbers must sum to 1.

```
(define frame:editor<%>
  (interface ()
    get-editor% ;; : → (implements editor<%>)
    get-editor<%>s ;; : → (listof interface)
    make-editor)) ;; : → (implements editor<%>)

(define frame:editor-mixin
  (mixin (frame<%>) (frame:editor<%>)
    (define/public (get-editor%)
      text%)

     (define/public (get-editor<%>s)
       (list editor<%>))

     (define/private (make-editor)
       (let ([editor% (get-editor%)])
         (let ([editor<%>s (get-editor<%>s)])
           (unless (andmap (λ (editor<%>) (implementation? editor% editor<%>))
                           editor<%>s)
             (error 'frame:editor%
                    "result of get-editor% must match ~e; got: ~e"
                    editor<%>s editor%)))
         (make-object editor%)))))
```

Figure 1.3: Editor Mixin Contract

### 1.2.2 Class-based Contracts

Beyond the standard features of Scheme, DrScheme's implementation language supports a class-based object system, similar to that provided by Java [20]. In addition to classes, interfaces, and objects supported by Java, DrScheme's class system supports mixins [17], which are class extensions parameterized over their superclass. Like classes, the body of a mixin consists of field and method declarations. Unlike classes, the programmer does not specify a superclass for a mixin. Instead, the programmer specifies an interface that each eventual superclass must implement. To build a class hierarchy, the programmer composes the mixins with classes and other mixins.

As an example mixin, consider the *frame:editor-mixin* in figure 1.3. This mixin extends classes that implement the *frame<%>* interface (that is, frames), and the mixin implements the *frame:editor<%>* interface. The body of the mixin supplies implementations of the methods in *frame:editor<%>*.

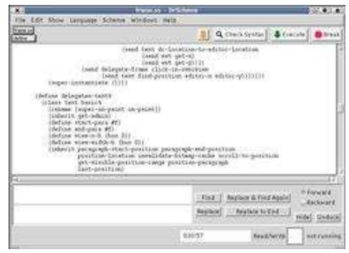
Together, methods in *frame:editor<%>* establish the connection between a frame and an editor that is visible in the frame. The *make-editor* method creates new editors. It uses the *get-editor%* method to determine the class for the editor and ensures that the instance of the class implements each of the interfaces returned by *get-editor<%>s*. The *get-editor%* method and the *get-editor<%>s* methods are intended to be overridden by extensions of this mixin. To extend the frame class with additional functionality that depends on the editor, a programmer overrides the *get-editor<%>s* method to return additional interfaces. Returning more interfaces in this fashion guarantees that the editor also supports the new functionality.

For two example frame mixins see figure 1.4. The *frame:searchable-mixin* provides an Emacs-like [21, 48] searching window in the bottom of the frame. The contour mixin provides a 20,000 foot overview of the program text, showing one pixel for each character in the definitions window. The two screen shots of DrScheme on the right-hand side of figure 1.4 show the DrScheme window with the searching mixin and the contour mixin.

Each mixin overrides the *get-editor<%>s* method, which guarantees that the frame's editor implements the appropriate interfaces. The *frame:searchable-mixin* requires the frame's editor to implement the *text:searching<%>* interface, so it can safely apply the *search* method. Similarly, the *frame:contour-mixin* requires the editor to implement the *text:contour<%>* interface, so it can safely invoke the *get-contour* method.

To construct a frame and an editor, a programmer composes a series of frame mixins and a series of editor mixins and overrides the frame's *get-editor%* method to return the editor mixin composition. The contracts ensure that the editor and the frame match each other. The first two expression in figure 1.5 shows how to compose the text mixins and frame mixins with the base text and frame classes. The final expression connects the frame class to the text class. Since the *my-text%* class implements the interfaces required by

```
(define frame:searchable<%>
  (interface ()
    show/hide-search-window))

(define text:searching<%>
  (interface ()
    search))

(define frame:searchable-mixin
  (mixin (frame:editor<%>)
         (frame:searchable<%>)
    (inherit get-editor)
    (define/override (get-editor<%>)
      (cons text:searching<%>
            (super get-editor<%>)))
    ··· (send (get-editor) search) ···))

(define frame:contour<%>
  (interface ()
     show/hide-contour-window))

(define text:contour<%>
  (interface ()
    get-contour))

(define frame:contour-mixin
  (mixin (editor<%>) (frame:contour<%>)
    (inherit get-editor)
    (define/override (get-editor<%>)
      (cons text:contour<%>
            (super get-editor<%>)))
    ··· (send (get-editor) get-contour ···) ···))
```



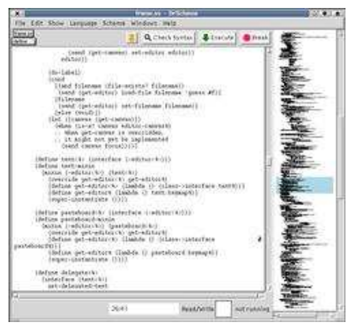


Figure 1.4: Editor Mixin Extensions

the mixin composition, no contract error is signaled for this composition. The screen shot on the right-hand side of figure 1.5 shows the resulting DrScheme window, with both the searching window and the contour window.

```
(define my-text%
  (text:searchable-mixin
    (text:delegate-mixin
     text%)))

(define super-frame%
  (frame:searchable-mixin
    (frame:delegate-mixin
      (frame:editor-mixin
       frame%))))

(define my-frame%
  (class super-frame%
    (define/override (get-editor%)
       my-text%)))
```

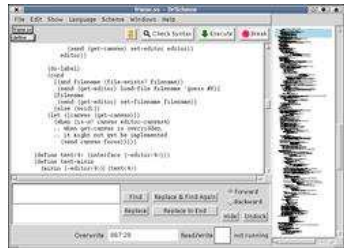


Figure 1.5: Mixin Composition

### 1.2.3 Higher-order Contracts

It is natural to expect contracts to express behaviors of functions in languages with higher-order functions. As an example, consider this predicate:

```
;; default-display-fraction? : number → boolean
(define (default-display-fraction? x)
  (and (number? x)
       (exact? x)
       (real? x)
       (not (integer? x))))
```

It controls one aspect of DrScheme's pretty-printer. If this predicate returns #t, the printer uses a graphical, mixed-notation fraction to display the value. If it returns #f, the printer uses a string of ASCII digits.

DrScheme is parameterized over this predicate. It allows tool-based extensions to control when these fractions are displayed, via the *set-display-fraction* and *current-display-fraction?* functions, as shown in figure 1.6. In order for the graphical display code to work properly, however, the argument to *set-display-fraction* must not return #t more often than *default-display-fraction?* predicate does, although it may return #f more often. Thus,

```
;; current-display-fraction? : number → boolean
(define current-display-fraction? default-display-fraction?)

;; set-display-fraction : (number → boolean) → void
(define (set-display-fraction f?)
  (set! current-display-fraction?
    (λ (num)
      (let ([curr (f? num)])
        (let ([def (default-display-fraction? num)])
          (when (and (not def) curr)
            (error 'set-display-fraction
                   "the predicate ~s returned #t when the default predicate did not"
                   f?)))
        curr))))
```

Figure 1.6: Controlling DrScheme's Printer

the *set-display-fraction* function's contract must guarantee that if its argument returns #t, *default-display-fraction?* would also have returned #t. The boxed code in the figure enforces this contract.

## 1.3 Thesis Statement and Roadmap

This dissertation investigates the use of contracts like those in the previous section, explaining how to enforce them at run-time and how to automatically assign blame when the contracts are violated.

The thesis of this dissertation is:

**Contract checking beyond procedural languages is complex and requires solid theoretical foundations.**

The dissertation supports the thesis with

- an examination of existing contract checkers for object-oriented languages and their failures to detect violations and assign blame properly,

- the design of a contract system that remedies the flaws in existing object-oriented contract checkers,
- the design of a contract system for higher-order contract checking, and
- contract soundness theorems that guarantee the contract checkers properly enforce contracts and properly assign blame for contract violations.

The dissertation is divided into 6 chapters. The first is this introduction.

Chapter 2 explains behavioral subtyping and how it should interact with contract checking. It examines the related work on contract checking and shows how all existing object-oriented contract checkers fail to check contracts properly.

Chapter 3 presents a contract compiler for the contract language discussed in chapter 2 that checks object-oriented contracts correctly. The compiler demonstrates that the contract checker must be integrated with the type-checker.

Chapter 4 provides the first step towards a theory of contract soundness, in analogy to the theory of type soundness. It formulates a contract soundness theorem as a relationship between a program running with contract checking enabled and the same program running without any contract checking. The theorem guarantees that if the program without any contract checking enters an invalid state, the checked program signals an appropriate error and provides the correct blame. Additionally, it guarantees that if the program without contract checking never violates a contract, the checked program and the unchecked program have identical behavior.

Chapter 5 presents the first contract checking calculus for languages with higher-order functions, shows how to implement it, and proves that the implementation matches the calculus. Chapter 6 concludes with a discussion of the dissertation's contributions and future work.

# Chapter 2

# Behavioral Subtyping and Related Work

Run-time enforced behavioral contracts have been studied extensively in the context of procedural languages [22, 37, 44, 47]. Rosenblum [47], in particular, makes the case for the use of assertions in C and describes the most useful classes of assertions.

Contract checking has also been added to many object-oriented languages [9, 19, 24, 28, 29, 38, 41, 45]. Even though these languages all support type hierarchies, none of their contract checkers take the hierarchies into account properly. In particular, the contracts on overriding methods are improperly synthesized from the programmer's original contracts. This flaw leads to mis-assigned, delayed, or even missing blame for contract violations.

This chapter is organized into five sections. The first explains behavioral contracts. The second explains behavioral subtyping. Section 2.3 explains the connection between behavioral contracts and behavioral subtyping. Section 2.4 demonstrates how existing contract checkers fail and section 2.5 shows how to check behavioral contracts in object-oriented languages properly.

## 2.1 Behavioral Contracts

.

In programs without subtyping, checking pre- and post-conditions is a simple matter. Consider this code, which implements a wrapper class for floats:

```
class Float {
  Float getValue() { ··· }
  Float sqrt() { ··· }
    @pre { getValue() > 0 }
    @post { Math.abs(@ret * @ret – this.getValue()) ≤ 0.1 }
}
```

In this case, the pre-condition for *sqrt* ensures that the method is only applied to positive numbers. The post-condition promises that the square of the result is within a certain tolerance of the original argument. We use @**ret** to stand for the result of a method in its post-condition.

In the case of the *sqrt* method, the pre- and post-conditions fully specify its correctness. In general, however, programmers do not use pre- and post-conditions to specify the entire behavior of the method; instead programmers use contracts to refine type specifications.

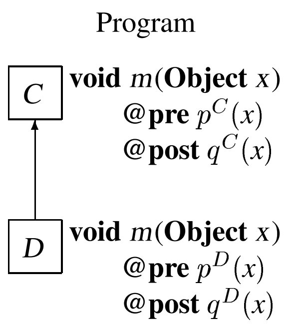
As long as programs do not use inheritance, contract checking is simply evaluating the conditions that the programmer stated. Once programs employ inheritance, contract monitoring requires more sophistication. In particular, subtypes of both classes and interfaces must be behavioral subtypes.

## 2.2 Behavioral Subtyping

Behavioral subtyping [1, 34, 35, 40] guarantees that all objects of a subtype preserve all of the original type's invariants. Put differently, any object of a subtype must be *substitutable* for an object of the original type without any effect on the program's observable behavior. For pre- and post-conditions, behavioral subtyping means that the pre-condition contracts for a type imply the pre-condition contracts for each subtype and the post-condition contracts for each subtype imply the post-condition contracts for the type.

Consider figure 2.1. It represents a class hierarchy fragment with two classes, $C$ and $D$, with $D$ derived from $C$. Both classes have a method $m$, with $D$'s $m$ overriding $C$'s $m$. Each method, however, has its own distinct pre-condition and post-condition. Interpreted for this fragment, the behavioral subtyping condition states that for each argument $x$ to the method, $p^C(x)$ implies $p^D(x)$ and $q^D(x)$ implies $q^C(x)$.

We generalize the behavioral subtyping condition to multiple inheritance by considering each subtype relationship independently. For an example, consider figure 2.2. It contains three interfaces, $L$, $R$, and $B$. Since each inheritance relationship is considered separately, we only require that $B$ is independently substitutable for either $L$ and $R$, as

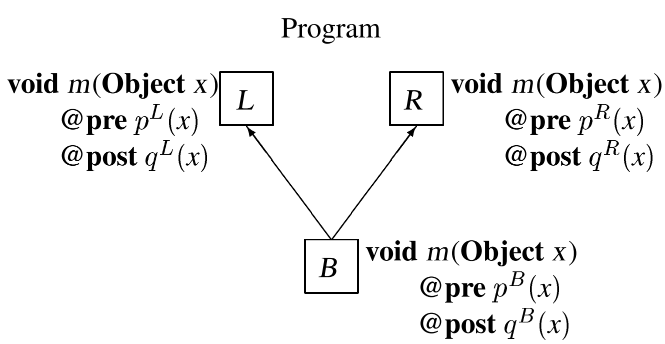



Conditions

$$\forall x\colon p^C(x) \Rightarrow p^D(x)$$

$$\forall x\colon q^D(x) \Rightarrow q^C(x)$$

Figure 2.1: The Behavioral Subtyping Condition




Conditions

$$(\forall x\colon p^R(x) \Rightarrow p^B(x)) \wedge (\forall x\colon p^L(x) \Rightarrow p^B(x))$$

$$(\forall x\colon q^B(x) \Rightarrow q^R(x)) \wedge (\forall x\colon q^B(x) \Rightarrow q^L(x))$$

Figure 2.2: The Behavioral Subtyping Condition, Generalized to Multiple Inheritance

reflected in the conditions listed in figure 2.2. This is the minimum requirement to match the spirit of the behavioral subtyping condition.*

## 2.3 Contracts and Behavioral Subtypes

A contract checker for object-oriented languages must verify that the pre-condition and post-condition hierarchies meet the behavioral subtyping requirement. This section ex-

*It is possible to imagine a stronger constraint, however. One may require that $L$ and $R$'s conditions be equivalent. This work applies for this stricter constraint, *mutatis mutandis*.

plains why with an example.

Consider this program:

**interface** *IConsole* {
  **int** *getMaxSize*();
    @**post** { @**ret** $> 0$ }
  **void** *display*(*String* $s$);
    @**pre** { $s.length() < this.getMaxSize()$ }
}

**class** *Console* **implements** *IConsole* {
  **int** *getMaxSize*() { $\cdots$ }
    @**post** { @**ret** $> 0$ }
  **void** *display*(*String* $s$) { $\cdots$ }
    @**pre** { $s.length() < this.getMaxSize()$ }
}

The *IConsole* interface contains the methods, types, and pre- and post-conditions for a small gas station console that displays messages to the gas pump operator. The *Console* class provides an implementation of *IConsole*. The *getMaxSize* method returns the limit on the message's size, and the *display* method changes the console's visible message. The post-condition for *getMaxSize* and the pre-condition for *display* merely guarantee some invariants of the console; they do not ensure correctness.

Consider this extension of *Console*:

**class** *RunningConsole* **extends** *Console* {
  **void** *display*(*String* $s$) {
    $\cdots$ *super.display*
        (*String.substring*
          ($s, \cdots, \cdots +$ *getMaxSize*())) $\cdots$
  }
    @**pre** { true }
}

The *display* method in this class creates a thread that displays whatever portion of the string fits in the console and then updates the console's display, scrolling the message character by character to display advertising messages while the gas is being pumped. Since the pre-condition of *display* in *RunningConsole* is true, it is implied by the pre-condition in *Console*, and thus *RunningConsole* is a behavioral subtype of *Console*.

Not every subtype is a behavioral subtype. Concretely, extensions of the *Console* class may have pre-conditions that are not implied by the supertype's pre-condition. Consider this example:

```
class PrefixedConsole extends Console {
  String getPrefix() {
    return ">> ";
  }
  void display(String s) {
    super.display(this.getPrefix() + s);
  }
    @pre { s.length() <
            this.getMaxSize() – this.getPrefix().length() }
}
```

The *PrefixedConsole* class provides debugging support for the console in the form of a prefix string that is attached to each message displayed in the console. The prefix string indicates the internal state of the gas pump.

Unlike *RunningConsole*, the pre-condition on *PrefixedConsole* is not implied by the pre-condition on *Console*. Accordingly, code that is written to accept instances of *Console* may violate the pre-condition of *PrefixedConsole* without violating the pre-condition of *Console*. Clearly, the code that expects instances of *Console* should not be blamed, since that code fulfilled its obligations by meeting *Console*'s pre-condition. Instead, the blame must lie with the programmer of *PrefixedConsole* for failing to create a behavioral subtype of *Console*.

In addition to classes, interfaces also describe a type hierarchy. Like the class type hierarchy, the interface type hierarchy must also specify a hierarchy of behavioral subtypes. Thus, unlike pre- and post-condition failures, the blame for a malformed hierarchy might fall on the author of code that contains only interfaces. Consider the two-part Java program in figure 2.3.

Imagine that two different programmers, Guy and James, wrote the two parts of the program. First, James's *main* method creates an instance of $C$, but with type $I$. Then, it invokes *m* with 5. According to the contracts for $I$, this is a perfectly valid argument. Ac-

```
// Written by Guy
interface I {
    void m(int a);
        @pre { a > 0 }
}


interface J extends I {
    void m(int a);
        @pre { a > 10 }
}
```

```
// Written by James
class C implements J {
    void m(int a) { ··· }
        @pre { a > 10 }

    public static void
    main(String argv[]) {
        I i = new C();
        i.m(5);
    }
}
```

Figure 2.3: Behavioral Subtyping in Interfaces

cording to the contract on $J$, however, this is an illegal argument. The behavioral subtyping condition tells us that $J$ can only be a subtype of $I$ if it is substitutable for $I$ in every context. This is not true, however, as $J$'s $m$ accepts fewer arguments than $I$'s $m$. In particular, $J$'s $m$ does not accept $1, 2, \cdots, 10$ but $I$'s $m$ does. Thus, Guy's claim that $J$ **extends** $I$ is wrong, with respect to the behavioral subtyping condition. When the method call from James's code fails, the blame for the contractual violation must lie with Guy.

The preceding examples suggest that contract checking systems for object-oriented languages should signal *three* kinds of errors: pre-condition violations, post-condition violations, and hierarchy errors. The latter distinguishes contract checking in the procedural world from contract checking in the object-oriented world. A hierarchy error signals that some subclass or extending interface is not a behavioral subtype, either because the hierarchy of pre-conditions or the hierarchy of post-conditions is malformed.

## 2.4 Problems with Prior Work

I examined six tools that implement Eiffel-style [41] contracts for Java: iContract [29], JPP [26], Kiev [27], JMSAssert [38], jContractor [24], and HandShake [9]. These systems enforce contracts by evaluating pre-condition expressions as methods are called and evaluating post-condition expressions as methods return. If the pre-condition fails, they blame the calling code for not establishing the proper context. If the post-condition fails, they blame the method itself for not living up to its promise.

The tools also handle programs with inheritance. With the exception of Jass, each tool constructs a disjunction of all of a method's super pre-conditions and a conjunction of all of the method's super post-conditions. For the program in figure 2.1, the systems replace the condition $p^D(x)$ with

$$p^C(x) \mid\mid p^D(x)$$

and replace the condition $q^D(x)$ with

$$q^C(x) \mathbin{\textbf{\&\&}} q^D(x)$$

Since the logical statements:

$$p^C(x) \Rightarrow (p^C(x) \mid\mid p^D(x))$$

and

$$(q^C(x) \mathbin{\textbf{\&\&}} q^D(x)) \Rightarrow q^D(x)$$

are always true, the re-written programs always satisfy the behavioral subtyping condition, *even if the original program did not*.

As Karaorman, Hölzle, and Bruno [24, section 4.1] point out, contract monitoring tools should check the programmer's original contracts, because checking the synthesized contracts can mask programmer errors. Recall the *PrefixedConsole* class from the previous section. That class's *display* method has the pre-condition contract:

$$\mathit{s.length}() < \mathit{this.getMaxSize}() - \mathit{this.getPrefix}().\mathit{length}()$$

and its superclass, *Console*, has a display method with this pre-condition contract:

$$\mathit{s.length}() < \mathit{this.getMaxSize}()$$

Since *Console*'s pre-condition does not always imply *PrefixedConsole*'s pre-condition, we know that the programmer of *PrefixedConsole* has made a mistake. The contract checking tool should, in turn, report this mistake to the programmer.

None of the existing tools for monitoring pre- and post-conditions handle this situation properly. Instead, they combine the two pre-conditions with a disjunction, and replacing $J$'s pre-condition with

$$\mathit{s.length}() < \mathit{this.getMaxSize}() \\ \textbf{or} \\ \underline{\mathit{s.length}() < \mathit{this.getMaxSize}() - \mathit{this.getPrefix}().\mathit{length}()}$$

As a result, the pre-condition on the overriding method (underlined above) is effectively ignored.

To see how bad the problem is, I translated *IConsole*, *Console* and *PrefixedConsole* to iContract syntax [29], using `4` as the maximum and a dummy *display* routine that just prints to stdout. The *main* method that invokes the *PrefixedConsole*'s *display* method with the string `"abc"`, as follows:

**new** *PrefixedConsole*().*display*(`"abc"`);

This call is erroneous, since the pre-condition on *PrefixedConsole*'s *display* method requires the argument to be a string of at most one character. iContract responded with this error message:[†]

```
java.lang.RuntimeException: error: precondition violated
(Console::display(String)):
(/*declared in IConsole::display(String)*/
(s.length() < this.getMaxSize()))
   at Console.display
   at PrefixedConsole.display
   at Main.main
```

That is, iContract blames the call to *super.display* inside *PrefixedConsole*'s *display* method, rather than blaming the hierarchy or the caller of *PrefixedConsole*'s *display*. In a larger program, such an erroneous explanation will send the programmer in the wrong direction when searching for the defect.

[†] iContract's response has been edited for clarity and formatting; the meaning is preserved.

Written by Alice

```
class C {
    void set(int a) { ··· }
        @pre { a > 0 }

    int get() { ··· }
        @post { a > 0 }
}
```

```
class D extends C {
    void set(int a) { ··· }
        @pre { a > 10 }

    int get() { ··· }
        @post { a > 10 }
}
```

Written by Bill

```
D d = new D() ;
d.set(5) ;
···
d.get() ;
```

Figure 2.4: Delayed, Incorrect Explanation for Contract Violation

In addition to blaming the wrong method calls, these erroneous contract monitoring systems may also trigger an exception much later than the contract violation actually occurs. Figure 2.4 contains a program fragment that illustrates this idea. It consists of two classes, $C$ and $D$, both of which implement an integer state variable. In $C$, the state variable is allowed to take on all positive values and in $D$, the state variable must be strictly larger than 10. Here, Alice wrote a hierarchy that does not match the behavioral subtyping condition, because $D$ is not a behavioral subtype of $C$. Since the existing tools combine the pre-conditions with a disjunction, $D$'s pre-condition is effectively the same as $C$'s and does not guarantee that the state variable is larger than 10. Thus, the call to *set* with 5 will not signal an error. Then, when *get* is invoked, it will return 5, which incorrectly triggers a post-condition violation, blaming Alice with an error message. Even though the blame is assigned to the guilty party in this case, it is assigned after the actual violation occurs (potentially even days later) and it is justified with an incorrect reason, decreasing the faith in the tools. This makes the problem both difficult to reproduce and to understand.

Existing contract monitoring systems handle Java's multiple inheritance for interfaces in a similarly flawed manner. When a single class implements more than one interface, JMSAssert [38] collects both the pre-conditions and post-conditions together in conjunctions, ensuring that the object meets all of the interfaces simultaneously. iContract [29] collects all of the pre-conditions in a disjunction and post-conditions in a conjunction. Again, since these manufactured contracts do not match the programmer's written contracts, blame for faulty programs may be delayed, mis-assigned, or entirely absent.

### 2.4.1 Jass

Jass [3, 4] is the only contract checker for Java that takes the contract hierarchy into consideration. To discover hierarchy errors with Jass, a programmer must specify a simulation method that creates an object of a supertype from the current object. The contract checker uses this simulation method to create a supertype object each time a method is called or a method returns. It checks that the relevant contracts of the supertype object and the original object are related via the proper implications. If not, the contract checker signals a hierarchy error.

Jass is based closely on Liskov and Wing's work [35]. It directly translates their framework to Java. Unfortunately, this framework and Java are mismatched in several ways. First, subtypes in Java must function on the same state space as their respective supertypes. This implies that the programmer should not have to define a simulation method and that the checks can be significantly cheaper than Jass's because no new objects need to be created. Second, this technique does not scale well to multiple inheritance. Third, this technique only checks a single step of subtype hierarchy, even though the type hierarchy in Java may have many steps.

## 2.5 Properly Monitoring Contracts

Programmers make mistakes. Their mistakes range from simple typographical errors to complex, subtle logical errors. Accordingly, tools must not be based on the assumption that programmers have constructed well-formed programs; in particular, tools should not re-write programs based on such assumptions. Instead, tools should report errors based on the program text that the programmers provide. Giving programmers good explanations in terms of their original programs helps them pinpoint their mistakes, in a precise and timely

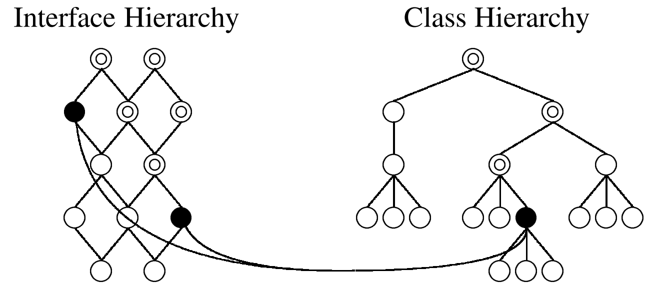



Figure 2.5: Hierarchy Checking

fashion. This is especially true for contract monitoring tools, whose purpose is to provide checkable specifications of programs to improve software reliability.

Consider the following code fragment in our running example of display consoles:

```
IConsole o = ConsoleFactory.newConsole();
// Assume that ConsoleFactory returns a PrefixedConsole
String s = "Falls in gap";
o.display(s);
```

When the *display* method is invoked, the pre-condition for *PrefixedConsole* fails. The blame, however, does not lie with the author of this code. Instead, the pre-conditions on *PrefixedConsole* and *Console* have the wrong relationship and the blame lies with the author of *PrefixedConsole* who implied that *PrefixedConsole* was a behavioral extension of *Console*. To assign blame correctly, contract checking tools should check for, and report, three different types of errors: pre-condition errors, post-condition errors, and *hierarchy extension* errors.

Checking hierarchy extension errors must be separate from checking pre- and post-condition errors, because the blame for the contract failure is not assigned to the same programmer. At method calls, in addition to checking the pre-condition of the method being invoked, the contract checking tool must verify that each pre-condition implies its subtype's pre-conditions, for the relevant portion of the contract hierarchy. Similarly at method returns, in addition to checking the post-condition of the returning method, the contract checking tool must verify that each post-condition implies its supertype's post-conditions, for the relevant portion of the contract hierarchy.

Figure 2.5 contains an example hierarchy. The right-hand side of the figure is the class hierarchy (tree) and the left-hand side is the interface hierarchy (dag). The curved lines indicate that the filled-in class implements the two filled-in interfaces. At each method call and return for instantiations of the filled-in class, the pre- and post-condition implications between the bullseye classes must be checked. I will show in the next chapter how to compile contracts so that hierarchy checking is performed in this manner.

# Chapter 3

# Contract Compilation

A contract monitor acts like a compiler. More concretely, it eliminates contracts from the program's source text and insert statements that validate the contracts at runtime. This process is best described as a compiler that consumes a language containing contract annotations and produces one without. This chapter develops such a contract compiler for Java that properly handles hierarchy violations.

The chapter is organized in three sections. The first describes the contract compiler with a series of examples. The second discusses how to integrate the compiler with Java's compilation model and the last discusses efficiency issues.

## 3.1 How to Check Contracts and Assign Blame

The contract compiler consumes a Java-like language with contract specifications on methods and interfaces, and produces plain Java, augmented with three new statements: **preBlame**, **postBlame**, and **hierBlame**. Each of these new statements accepts a string naming the class that is to be blamed for the respective failure. When they are executed, the program halts with an appropriate error message that blames the author of the class named by the argument.

Abstractly, the contract compiler transfers pre-condition and post-condition contracts into wrapper methods that check the contracts and call the corresponding original method. It rewrites calls to methods with contracts into calls to the appropriate wrapper method. Furthermore, method calls in the elaborated program are rewritten to call these wrapper methods, based on the static type of the object whose method is invoked. Thus, the translation depends on the type analysis and takes into account the type hierarchy. As such, the

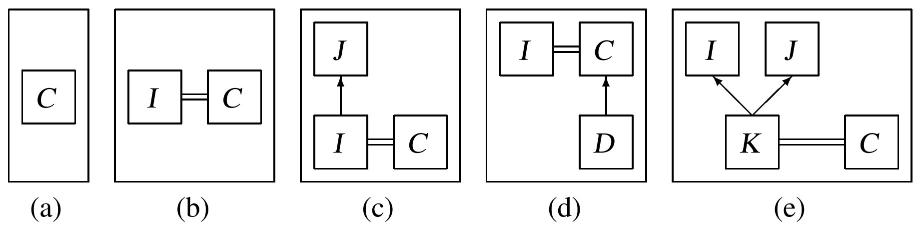



Figure 3.1: Section 3.1 Overview

contract compiler must be integrated with the type-checker.

Figure 3.1 shows a series of hierarchy diagrams that provide an outline for this section. Each diagram corresponds to a configuration of classes and interfaces. The boxes represent classes and interfaces. The classes are named $C$ and $D$ and the interfaces are named $I$, $J$, and $K$. The single lines with arrow-heads represent both class and interface inheritance and the double lines (without arrow-heads) represent interface implementation.

### 3.1.1 Flat Contract Checking

**Diagram 3.1 (a)** illustrates the simplest case. Figure 3.2 contains program text corresponding to this diagram and its translation. The program consists of two classes, $C$ and *Main*. The original $C$ class has a method $m$ with a pre-condition and a post-condition. Its translation has two methods, the original $m$ and the wrapper method $m\_C$. The name of the wrapper method is synthesized from the name of the original method and the name of the class. The wrapper method accepts one additional argument naming the class that is calling the method, which is blamed if the pre-condition fails. In figure 3.2 lines 4–6, the wrapper method checks the pre-condition and blames the class of the caller if a violation occurs. Then, in line 7, it runs the original method. Finally, in lines 8–10, it checks the post-condition, blaming the class itself for any violations of the post-condition. The contract compiler also rewrites the call to $m$ in *Main* to call the wrapper method, passing in

```
class C {
  void m(int a) { ··· }
    @pre
      ··· C's pre-condition ···
    @post
      ··· C's post-condition ···
}

class Main {
  public static void
  main(String[] argv) {
    new C().m(5) ;
  }}
```

```
class C {
  void m (int a) { ··· }
  void m_C (string tbb, int a) {
    if (! ··· C's pre-condition ···) {
      preBlame(tbb) ;
    }
    m(a) ;
    if (! ··· C's post-condition ···) {
      postBlame("C") ;
    }}}

class Main {
  public static void
  main(String[] argv) {
    new C().m_C("Main", 5) ;
  }}
```

Figure 3.2: Pre- and Post-condition Checking

"Main" as an additional argument to be blamed for a pre-condition violation.

### 3.1.2 Interface Implementation

**Diagram 3.1 (b)** contains a class and an interface. The class implements the interface. As with the previous example, when a method is called, its pre-condition must be checked, though the pre-condition to be checked depends on the static type of the reference to the object whose method is invoked. Since that may be either $I$ or $C$, two wrapper methods are generated: $m\_C$ and $m\_I$, which each check their respective pre- and post-conditions.

The example in diagram (b) adds another twist to the simple translation in figure 3.2. Since instances of $C$ are substitutable in contexts expecting $I$s, we must also check that the hierarchy is well-formed. In this case, $I$'s pre-conditions must imply $C$'s pre-conditions and $C$'s post-conditions must imply $I$'s post-conditions, for each method call to $m$. There are four possibilities for $C$ and $I$'s pre-conditions. Clearly, if both are true, no violation has occurred and if both are false, the pre-condition does not hold and the caller must be blamed. If $I$'s pre-condition is true and $C$'s pre-condition is false, the hierarchy is malformed and the author of $C$ must be blamed. If $I$'s pre-condition is false and $C$'s pre-condition is true, the hierarchy is well-formed and no hierarchy violation is signaled. In this case, however, if the object is being viewed as an instance of $I$, the pre-condition checking code in *m_I* blames the caller for failing to establish the pre-condition. If the object is being viewed as an instance of $C$, no error occurs and no violation is signaled. The logic of post-condition checking is similar.

To perform the hierarchy checks, hierarchy checking methods are generated for each interface and class method. For classes, the new methods are inserted into the translated version of the class. For interfaces, the new methods are inserted into a new class that is generated for each interface. These hierarchy checking methods recursively combine the result of each pre- or post-condition with the rest of the pre- and post-condition results in the hierarchy to determine if the hierarchy is well-formed.

Figure 3.3 contains a translation that illustrates how our compiler deals with diagram (b). The wrapper methods, *m_C* and *m_I*, are augmented with calls to the hierarchy checking methods, *m_pre_hier* and *m_post_hier* in figure 3.3 lines 11 and 16. The *m_pre_hier* and *m_post_hier* methods in $C$ ensure that the pre- and post-condition hierarchies are well-formed. The checkers for $I$ would appear in the *I_checkers* class; they are analogous and omitted.

For the pre-condition checking *m_pre_hier* accepts the same arguments as the original method and returns the value of the pre-condition. To check the hierarchy, the method first calls *I_checkers*'s *m_pre_hier* method in line 20, which ensures that the pre-condition hierarchy from $I$ (and up) is well formed. Since **this** in *I_checkers* does not refer to the object whose contracts are checked, the current object is passed along to *I_checkers*'s *m_pre_hier*. In our example, the hierarchy from $I$ (and up) is trivially well-formed, since $I$ has no supertypes. The result of *I_checkers*'s *m_pre_hier* is the value of $I$'s pre-condition on *m* and is bound to *sup* in $C$'s *m_pre_hier*, as shown on line 19. Then, *m_pre_hier* binds *res* to the value of its own pre-condition, in line 21. Next, it tests if $I$'s pre-condition implies $C$'s

```
interface I {
    void m(int a) ;
        @pre
            ··· C's pre-condition
        @post
            ··· C's post-condition
}

class C implements I {
    void m(int a) { ··· }
        @pre
            ··· I's pre-condition
        @post
            ··· I's post-condition
}
```

```
1: interface I { ··· }
2: class I_checkers { ··· }

4: class C implements I {
5:     void m () { ··· }
6:     void m_I (string tbb, int a) { ··· }
7:     void m_C (string tbb, int a) {
8:         if (! ··· C's pre-condition ···) {
9:             preBlame(tbb) ;
10:        }
11:        m_pre_hier(a) ;
12:        m(a) ;
13:        if (! ··· C's post-condition ···) {
14:            postBlame("C") ;
15:        }
16:        m_post_hier("C", false, a) ;
17:    }
18:    boolean m_pre_hier(int a) {
19:        boolean sup =
20:            I_checkers.m_pre_hier(this, a) ;
21:        boolean res = ··· C's pre-condition ;
22:        if (!sup || res) {            // sup ⇒ res
23:            return res ;
24:        } else {
25:          hierBlame("C") ;
26:        }}
27:    void m_post_hier(string tbb, boolean last, int a) {
28:        boolean res = ··· C's post-condition ;
29:        if (!last || res) {           // last ⇒ res
30:            I_checkers.m_post_hier ("C", res, this, a) ;
31:        } else {
32:          hierBlame(tbb) ;
33:        }}}
```

Figure 3.3: Hierarchy Checking

pre-condition, with the expression *!sup* || *res* in line 22, which is logically equivalent to *sup* $\Rightarrow$ *res*. If the implication holds, *m_pre_hier* returns the result of the pre-condition, in line 23. If not, it evaluates the **hierBlame** statement in line 25, which aborts the program and blames $C$ as a bad extension of $I$.

The post-condition checking recursively traverses the interface and class hierarchy in the same order as pre-condition checking. In contrast to the pre-condition checking, post-condition checking accumulates the intermediate results needed to check the hierarchy instead of returning them. In our example, the first two arguments to *m_post_hier* in $C$ are the accumulators: *tbb* (figure 3.3 line 27) is the class to be blamed for the failure and *last* is the value of the post-condition of a subtype (initially false because there are no relevant subtypes). To determine if there is a hierarchy violation, *res* is bound to the value of $m$'s post-condition in line 28, and the implication is checked in line 29. If the hierarchy is flawed at this point, *tbb* is blamed in line 32. In this example, this cannot happen, since *res* is initially false, but the code is needed in general. Then, *I_checkers*'s *m_post_hier* is called in line 30, with the value of $C$'s post-condition and $C$'s name. Thus, the blame for a bad hierarchy discovered during *I_checkers*'s *m_post_hier* falls on $C$.

**Diagram 3.1 (c)** adds interface checking to the picture. In principle, the contract checkers for the program in diagram (c) are the same as those in diagram (b). The additional interface generates an additional class for checking the additional level in the hierarchy, but otherwise contract checking proceeds as in diagram (b).

### 3.1.3 Class Inheritance

**Diagram 3.1 (d)** introduces class inheritance (or implementation inheritance). It poses a more complex problem for our contract compiler. As with an additional interface, new methods are generated to check the hierarchy. The new hierarchy checking methods, however, are only used when an instance of the derived class is created. That is, if the program creates only instances of $C$, the hierarchy below $C$ is not checked. Instances of $D$, however, do check the entire hierarchy, including $C$'s and $I$'s pre- and post-conditions. In general, the conditions of every interface and every superclass of the originally instantiated class are checked at each method call and each method return to ensure the hierarchy is sound.

### 3.1.4 Multiple Inheritance

**Diagram 3.1 (e)** shows an interface with two super-interfaces. According to the discussion in chapter 2, the hierarchy checkers must check that the pre-condition in $I$ implies the pre-condition in $K$ and the pre-condition in $J$ implies the pre-condition in $K$.[*] The following

[*] Another alternative, as mentioned in footnote $*$ in chapter 2, is to ensure that $I$'s and $J$'s conditions are equivalent. This could easily be checked at this point in the hierarchy checker.

```
interface I {
    void m(int a);
        @pre
          ··· I's pre-condition ···
        @post
          ··· I's post-condition ···
}

interface J {
    void m(int a);
        @pre
          ··· J's pre-condition ···
        @post
          ··· J's post-condition ···
}

interface K extends I, J {
    void m(int a);
        @pre
          ··· K's pre-condition ···
        @post
          ··· K's post-condition ···
}
```

```
1: class I_checkers { ··· }
2: class J_checkers { ··· }

4: class K_checkers {
5:     static boolean m_pre_hier(K this, int a) {
6:         boolean sup =
7:                 I_checkers.m_pre_hier(this, a) ||
8:                 J_checkers.m_pre_hier(this, a) ;
9:         boolean res = ··· K's pre-condition ··· ;
10:        if (!sup || res) {              // sup ⇒ res
11:            return res ;
12:        } else {
13:          hierBlame("K") ;
14:        }}
15:    static void m_post_hier(string tbb, boolean last,
16:                            K this, int a) {
17:        boolean res = ··· K's post-condition ··· ;
18:        if (!last || res) {             // last ⇒ res
19:            return
20:                I_checkers.m_post_hier("K", res, this, a)
21:                &&
22:                J_checkers.m_post_hier("K", res, this, a);
23:        } else {
24:          hierBlame(tbb) ;
25:        }}}
```

Figure 3.4: Hierarchy Checking for Multiple Inheritance

boolean identity

$$(a \to c) \land (b \to c) \Leftrightarrow (a \lor b) \to c$$

tells us that we can just check that disjunction of $I$'s and $J$'s pre-conditions implies $K$'s pre-condition. Accordingly, as shown in figure 3.4, *K_checkers*'s *m_pre_hier* method hierarchy checker combines the results of *I_checkers*'s and *J_checkers*'s *m_pre_hier* methods in a disjunction and binds that to *sup* in figure 3.4 lines 6–8. Thus, the contract checker's traversal of the type hierarchy remains the same.

For post-conditions, we take advantage of a similar boolean identity:

$$(a \to b) \land (a \to c) \Leftrightarrow a \to (b \land c)$$

and combine the recursive calls with a conjunction to compute the result of the post-condition hierarchy checking method, as shown in *m_post_hier*'s definition in figure 3.4 lines 19–22.

## 3.2 Environmental Considerations

Our contract compiler does not need to install a class loader, generate any extra `.java` or `.class` files, nor does it require any extra class libraries during evaluation. These features of our design enable our contract compiler to integrate seamlessly with Java's compilation model, unlike other existing Java contract checkers.

This smooth interaction is due to the fact that the compiler produces a single Java `.class` file for each source class and interface. As described in the previous section, however, the contract compiler generates an additional class for each interface. An implementation should instead augment the `.class` file generated for the interface with enough information to add the wrapper and hierarchy methods to each class that implements the interface. This would be done using a custom attribute in the class file that contains the byte-codes of the contracts. The hierarchy checking methods for interfaces are then copied into classes that implement interfaces. This would require some code duplication, but it would not be much, for most programs.

In the code examples in section 3.1, we used method names for wrappers that are valid Java identifiers. In an implementation, special names for wrapper methods could be used in the class files to eliminate name clashes with programmer-defined method names. Additionally, an implementation would not add new blame-assigning statements to Java; instead it would inline code that raises an exception to blame the guilty party.

Since our contract compiler uses wrapper methods to check contracts and redirects each method call to call the wrapper methods, the programmer's original methods are still available in the class. Thus, the `.class` files that our contract compiler generates can be linked to existing, pre-compiled byte-codes. This allows pre-existing byte-code distributions of Java code to interoperate with code compiled by our contract compiler, but at the cost of losing contract enforcement. Since existing byte-code libraries would bypass the wrapper methods and call the original methods directly, no pre-condition, post-condition or hierarchy checking contract checking occurs.

## 3.3 Performance

Although our compiler design may seem far more expensive than traditional checkers, due to the hierarchy checking, it is not. The traditional contract checkers also combine each contract with the corresponding contracts in the supertype. Hence, both approaches to contract checking evaluate the same contracts at each method call. For post-condition checking, both approaches check the same contracts when the contracts succeed. For pre-condition checking, traditional checkers might possibly short-cut some of the checks that our contract checker would check (which leads to the mis-assigned blame explained earlier). This additional cost thus pays for the additional guarantees of our approach.

Additionally, the method chaining in our hierarchy checking methods is unnecessary. Because the class and interface hierarchies are static, the method chaining could be replaced with nested **if** tests. If done in a naive manner, however, this could lead to code explosion. An optimizing Java compiler could, however, inline the hierarchy checking method calls when it would be profitable to do so.

# Chapter 4

# Contract Soundness

The failure of existing tools to properly assign blame for contract violations can be traced directly to the lack of fundamental research on contract checking. To address this problem, this chapter develops a formal model of Java with contracts in the form of a calculus. The starting point is the Classic Java calculus [16]. It specifies the syntax, type system, and semantics of a small Java-like language. I extended it with mechanisms for simple pre- and post-condition contract specifications. Based on the extended calculus, contract checking is modeled as a translation from the extended language with contracts into the Class Java calculus extended with simple error-signalling primitives. Using the calculus, we state and prove a contract soundness theorem. This theorem guarantees that the contract compiler correctly catches contract violations and correctly assigns blame for contract violations.

This chapter is divided into six sections. The first section introduces the contract checking framework and shows how it is based on the type soundness framework. Section 4.2 presents the syntax and section 4.3 presents the type checker. Section 4.4 presents the contract elaborator. Section 4.5 presents the operational semantics. Finally section 4.6 states and proves the contract soundness theorem, for this contract elaborator.

## 4.1 From Type Soundness to Contract Soundness

The development of meaningful type systems has benefited from a well-developed theory [42]. In particular, good type systems satisfy a type soundness theorem, which ensures that the type checker respects the language's semantics and specifies what kinds of runtime errors a program may signal.

A type soundness theorem has two parts. First, it specifies what kind of errors (or run-time exceptions) the evaluation of a well-typed program is allowed to trigger. Second, it implies that certain properties hold for the evaluation of well-typed subexpressions. For example, an addition operation in an ML program will always receive two numbers, and thus ML programs never terminate with errors due to the misuse of the addition operation. An array indexing operation will always receive an integer as an index, but the integer may be out of the array's range. Hence, an ML program will never terminate with a non-integer used as an array index, but it may terminate due to an out of bounds array index.

We can show that a contract checking system can satisfy a contract soundness theorem. Like a type soundness theorem, the contract soundness theorem has two parts. First, it states what kind of errors the evaluation of a monitored program may signal. Second, it guarantees that the specified hierarchy of interfaces and classes satisfies implications between the stated pre- and post-conditions of overridden methods.

## 4.2 Syntax

Figure 4.1 contains the syntax for Contract Java (adapted from Flatt et al [16]). The syntax is divided into three parts. Programmers use syntax (a) to write their programs. The type checker elaborates syntax (a) to syntax (b), which contains type annotations for use by the evaluator and contract compiler. The contract compiler elaborates syntax (b) to syntax (c). It elaborates the pre- and post-conditions into monitoring code; the result is accepted by the evaluator.

A program *P* is a sequence of class and interface definitions followed by an expression that represents the body of the *main* method. Each class definition consists of a sequence of field declarations followed by a sequence of method declarations and their contracts. An interface consists of method specifications and their contracts. The contracts are arbitrary Java expressions that have type **boolean**.* A method body in a class can be **abstract**,

* We could have carried out our study in a more complex contract specification language, but plain Java expressions suffice to express many important contracts. Additionally, using a single language for pre-conditions, post-conditions, and expressions simplifies the presentation and proofs.

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ e \\
\\
\mathit{defn} & ::= & \textbf{class}\ c\ \textbf{extends}\ c \\
 & & \quad \textbf{implements}\ i^* \\
 & & \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
 & \mid & \textbf{interface}\ i\ \textbf{extends}\ i^* \\
 & & \{\ \mathit{imeth}^*\ \} \\
\\
\mathit{field} & ::= & t\ \mathit{fd} \\
\mathit{meth} & ::= & t\ \mathit{md}\ (\ \mathit{arg}^*\ )\ \{\ \mathit{body}\ \} \\
 & & \textbf{@pre}\ \{\ e\ \}\ \textbf{@post}\ \{\ e\ \} \\
\mathit{imeth} & ::= & t\ \mathit{md}\ (\ \mathit{arg}^*\ ) \\
 & & \textbf{@pre}\ \{\ e\ \}\ \textbf{@post}\ \{\ e\ \} \\
\mathit{arg} & ::= & t\ \mathit{var} \\
\mathit{body} & ::= & e \mid \textbf{abstract} \\
\\
e & ::= & \textbf{new}\ c \mid \mathit{var} \mid \textbf{null} \\
 & \mid & e.\mathit{fd} \mid e.\mathit{fd} = e \\
 & \mid & e.\mathit{md}\ (e^*) \\
 & \mid & \textbf{super}.\mathit{md}\ (e^*) \\
 & \mid & \textbf{view}\ t\ e \\
 & \mid & \textbf{let}\ \{\ \mathit{binding}^*\ \}\ \textbf{in}\ e \\
 & \mid & \textbf{if}\ (\ e\ )\ e\ \textbf{else}\ e \mid \textbf{true} \mid \textbf{false} \\
 & \mid & \{\ e\ ;\ e\ \} \\
\\
\mathit{binding} & ::= & \mathit{var} = e \\
\mathit{var} & ::= & \text{a variable name or } \mathit{this} \\
c & ::= & \text{a class name or } \textbf{Object} \\
i & ::= & \text{interface name or } \mathsf{Empty} \\
\mathit{fd} & ::= & \text{a field name} \\
\mathit{md} & ::= & \text{a method name} \\
t & ::= & c \mid i \mid \textbf{boolean}
\end{array}
$$

(a) Surface Syntax

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ e \\
\\
\mathit{defn} & ::= & \textbf{class}\ c\ \textbf{extends}\ c \\
 & & \quad \textbf{implements}\ i^* \\
 & & \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
 & \mid & \textbf{interface}\ i\ \textbf{extends}\ i^* \\
 & & \{\ \mathit{imeth}^*\ \} \\
\\
\mathit{field} & ::= & t\ \mathit{fd} \\
\mathit{meth} & ::= & t\ \mathit{md}\ (\ \mathit{arg}^*\ )\ \{\ \mathit{body}\ \} \\
 & & \textbf{@pre}\ \{\ e\ \}\ \textbf{@post}\ \{\ e\ \} \\
\mathit{imeth} & ::= & t\ \mathit{md}\ (\ \mathit{arg}^*\ ) \\
 & & \textbf{@pre}\ \{\ e\ \}\ \textbf{@post}\ \{\ e\ \} \\
\mathit{arg} & ::= & t\ \mathit{var} \\
\mathit{body} & ::= & e \mid \textbf{abstract} \\
\\
e & ::= & \textbf{new}\ c \mid \mathit{var} \mid \textbf{null} \\
 & \mid & e\underline{\ :\ c\ }.\mathit{fd} \mid e\underline{\ :\ c\ }.\mathit{fd} = e \\
 & \mid & e\underline{:\ t}.\mathit{md}\ (e^*) \\
 & \mid & \textbf{super}\underline{\ \equiv \mathit{this} : c\ }.\mathit{md}\ (e^*\ ) \\
 & \mid & \textbf{view}\ t\ e \\
 & \mid & \textbf{let}\ \{\ \mathit{binding}^*\ \}\ \textbf{in}\ e \\
 & \mid & \textbf{if}\ (\ e\ )\ e\ \textbf{else}\ e \mid \textbf{true} \mid \textbf{false} \\
 & \mid & \{\ e\ ;\ e\ \} \\
\\
\mathit{binding} & ::= & \mathit{var} = e \\
\mathit{var} & ::= & \text{a variable name or } \mathit{this} \\
c & ::= & \text{a class name or } \textbf{Object} \\
i & ::= & \text{interface name or } \mathsf{Empty} \\
\mathit{fd} & ::= & \text{a field name} \\
\mathit{md} & ::= & \text{a method name} \\
t & ::= & c \mid i \mid \textbf{boolean}
\end{array}
$$

(b) Typed Contract Syntax

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ e \\
\\
\mathit{defn} & ::= & \textbf{class}\ c\ \textbf{extends}\ c \\
 & & \quad \textbf{implements}\ i^* \\
 & & \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
 & \mid & \textbf{interface}\ i\ \textbf{extends}\ i^* \\
 & & \{\ \mathit{imeth}^*\ \} \\
\\
\mathit{field} & ::= & t\ \mathit{fd} \\
\mathit{meth} & ::= & t\ \mathit{md}\ (\ \mathit{arg}^*\ )\ \{\ \mathit{body}\ \} \\
\\
\mathit{imeth} & ::= & t\ \mathit{md}\ (\ \mathit{arg}^*\ ) \\
\\
\mathit{arg} & ::= & t\ \mathit{var} \\
\mathit{body} & ::= & e \mid \textbf{abstract} \\
\\
e & ::= & \textbf{new}\ c \mid \mathit{var} \mid \textbf{null} \\
 & \mid & e : c\ .\mathit{fd} \mid e : c\ .\mathit{fd} = e \\
 & \mid & e : t.\mathit{md}\ (e^*) \\
 & \mid & \textbf{super} \equiv \mathit{this} : c\ .\mathit{md}\ (e^*\ ) \\
 & \mid & \textbf{view}\ t\ e \\
 & \mid & \textbf{let}\ \{\ \mathit{binding}^*\ \}\ \textbf{in}\ e \\
 & \mid & \textbf{if}\ (\ e\ )\ e\ \textbf{else}\ e \mid \textbf{true} \mid \textbf{false} \\
 & \mid & \{\ e\ ;\ e\ \} \\
 & \mid & \textbf{return} : t, c\ \{\ e\ \} \\
 & \mid & \textbf{preErr}(e) \\
 & \mid & \textbf{postErr}(e) \\
 & \mid & \textbf{hierErr}(e) \\
\\
\mathit{binding} & ::= & \mathit{var} = e \\
\mathit{var} & ::= & \text{a variable name or } \mathit{this} \\
c & ::= & \text{a class name or } \textbf{Object} \\
i & ::= & \text{interface name or } \mathsf{Empty} \\
\mathit{fd} & ::= & \text{a field name} \\
\mathit{md} & ::= & \text{a method name} \\
t & ::= & c \mid i \mid \textbf{boolean}
\end{array}
$$

(c) Core Syntax

Figure 4.1: Contract Java syntax; before and after contracts are compiled away

The sets of names for variables, classes, interfaces, fields, and methods are assumed to be mutually distinct. The meta-variable $T$ is used for method signatures ($t \ldots \longrightarrow t$), $V$ for variable lists ($var \ldots$), and $\Gamma$ for environments mapping variables to types. Ellipses on the baseline ($\ldots$) indicate a repeated pattern or continued sequence, while centered ellipses ($\cdots$) indicate arbitrary missing program text (not spanning a class or interface definition).

| | |
|---|---|
| ClassesOnce($P$) | Each class name is declared only once |
| | **class** $c \cdots$ **class** $c' \cdots$ is in $P \implies c \neq c'$ |
| FieldOncePerClass($P$) | Field names in each class declaration are unique |
| | **class** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$ |
| MethodOncePerClass($P$) | Method names in each class declaration are unique |
| | **class** $\cdots \{ \cdots md\,(\cdots)\,\{\cdots\} \cdots md'\,(\cdots)\,\{\cdots\} \cdots \}$ is in $P \implies md \neq md'$ |
| InterfacesOnce($P$) | Each interface name is declared only once |
| | **interface** $i \cdots$ **interface** $i' \cdots$ is in $P \implies i \neq i'$ |
| MethodArgsDistinct($P$) | Each method argment name is unique |
| | $md\,(t_1\ var_1 \ldots t_n\ var_n)\,\{\cdots\}$ is in $P \implies var_1, \ldots var_n$, and *this* are distinct |
| $\prec^c_P$ | Class is declared as an immediate subclass |
| | $c \prec^c_P c' \Leftrightarrow$ **class** $c$ **extends** $c' \cdots \{\cdots\}$ is in $P$ |
| $\Subset^c_P$ | Field is declared in a class |
| | $\langle c.fd, t\rangle \Subset^c_P c \Leftrightarrow$ **class** $c \cdots \{ \cdots t\ fd \cdots \}$ is in $P$ |
| $\Subset^c_P$ | Method is declared in class |
| | $\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \Subset^c_P c$ $\Leftrightarrow$ **class** $c \cdots \{ \cdots t\ md\,(t_1\ var_1 \ldots t_n\ var_n)\,\{e\} \cdots \}$ is in $P$ |
| $\prec^i_P$ | Interface is declared as an immediate subinterface |
| | $i \prec^i_P i' \Leftrightarrow$ **interface** $i$ **extends** $\cdots i' \cdots \{\cdots\}$ is in $P$ |
| $\Subset^i_P$ | Method is declared in an interface |
| | $\langle md, (t_1 \ldots t_n \longrightarrow t)\rangle \Subset^i_P i$ $\Leftrightarrow$ **interface** $i \cdots \{ \cdots t\ md : (t_1\ arg_1) \ldots (t_n\ arg_n)$ **@pre** $\{ e_b \}$ **@post** $\{ e_a \} \cdots \}$ is in $P$ |
| $\prec\!\!\prec^c_P$ | Class declares implementation of an interface |
| | $c \prec\!\!\prec^c_P i \Leftrightarrow$ **class** $c \cdots$ **implements** $\cdots i \cdots \{\cdots\}$ is in $P$ |
| $\leq^c_P$ | Class is a subclass |
| | $\leq^c_P \equiv$ the transitive, reflexive closure of $\prec^c_P$ |
| CompleteClasses($P$) | Classes that are extended are defined |
| | $\mathrm{rng}(\prec^c_P) \subseteq \mathrm{dom}(\prec^c_P) \cup \{$**Object**$\}$ |
| WellFoundedClasses($P$) | Class hierarchy is an order |
| | $\leq^c_P$ is antisymmetric |
| ClassMethodsOK($P$) | Method overriding preserves the type |
| | $(\langle md, T, V, e\rangle \Subset^c_P c$ and $\langle md, T', V', e'\rangle \Subset^c_P c') \implies (T = T'$ or $c \not\leq^c_P c')$ |
| $\in^c_P$ | Field is contained in a class |
| | $\langle c'.fd, t\rangle \in^c_P c$ $\Leftrightarrow \langle c'.fd, t\rangle \Subset^c_P c'$ and $c' = \min\{c'' \mid c \leq^c_P c''$ and $\exists t'$ s.t. $\langle c''.fd, t'\rangle \Subset^c_P c''\}$ |
| $\in^c_P$ | Method is contained in a class |
| | $\langle md, T, V, e\rangle \in^c_P c$ $\Leftrightarrow (\langle md, T, V, e\rangle \Subset^c_P c'$ and $c' = \min\{c'' \mid c \leq^c_P c''$ and $\exists e', V'$ s.t. $\langle md, T, V', e'\rangle \Subset^c_P c''\})$ |

Figure 4.2: Predicates and relations in the model of Contract Java , i

| | |
|---|---|
| $\leq^{\mathsf{i}}_{P}$ | Interface is a subinterface<br>$\leq^{\mathsf{i}}_{P} \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{i}}_{P}$ |
| $\textsc{CompleteInterfaces}(P)$ | Extended/implemented interfaces are defined<br>$\mathrm{rng}(\prec^{\mathsf{i}}_{P}) \cup \mathrm{rng}(\prec\!\!\!\ll^{\mathsf{c}}_{P}) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_{P}) \cup \{\mathsf{Empty}\}$ |
| $\textsc{WellFoundedInterfaces}(P)$ | Interface hierarchy is an order<br>$\leq^{\mathsf{i}}_{P}$ is antisymmetric |
| $\ll^{\mathsf{c}}_{P}$ | Class implements an interface<br>$c \ll^{\mathsf{c}}_{P} i \Leftrightarrow \exists c', i'$ s.t. $c \leq^{\mathsf{c}}_{P} c'$ and $i' \leq^{\mathsf{i}}_{P} i$ and $c' \prec\!\!\!\ll^{\mathsf{c}}_{P} i'$ |
| $\textsc{InterfaceMethodsOK}(P)$ | Interface inheritance or redeclaration of methods is consistent<br>$\langle md, T\rangle \in\!\!\!\in^{\mathsf{i}}_{P} i$ and $\langle md, T'\rangle \in\!\!\!\in^{\mathsf{i}}_{P} i'$<br>$\Longrightarrow (T = T'$ or $\forall i''(i'' \not\leq^{\mathsf{i}}_{P} i$ or $i'' \not\leq^{\mathsf{i}}_{P} i'))$ |
| $\in^{\mathsf{i}}_{P}$ | Method is contained in an interface<br>$\langle md, T\rangle \in^{\mathsf{i}}_{P} i \Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_{P} i'$ and $\langle md, T\rangle \in\!\!\!\in^{\mathsf{i}}_{P} i'$ |
| $\textsc{ClassesImplementAll}(P)$ | Classes supply methods to implement interfaces<br>$c \prec\!\!\!\ll^{\mathsf{c}}_{P} i \Longrightarrow (\forall md, T\ \langle md, T\rangle \in^{\mathsf{i}}_{P} i \Longrightarrow \exists e, V'$ s.t. $\langle md, T, V', e\rangle \in^{\mathsf{c}}_{P} c)$ |
| $\textsc{NoAbstractMethods}(P, c)$ | Class has no **abstract** methods (can be instantiated)<br>$\langle md, T, V, e\rangle \in^{\mathsf{c}}_{P} c \Longrightarrow e \neq$ **abstract** |
| $\leq_{P}$ | Type is a subtype<br>$\leq_{P} \equiv \leq^{\mathsf{c}}_{P} \cup \leq^{\mathsf{i}}_{P} \cup \ll^{\mathsf{c}}_{P}$ |
| $\prec_{P}$ | Type is an immediate subtype<br>$\prec_{P} \equiv \prec^{\mathsf{c}}_{P} \cup \prec^{\mathsf{i}}_{P} \cup \prec\!\!\!\ll^{\mathsf{c}}_{P}$ |
| $\in_{P}$ | Field or Method is in a type (method/interface)<br>$\langle md, T\rangle \in_{P} i \Leftrightarrow \langle md, T\rangle \in^{\mathsf{i}}_{P} i$ |
| $\in_{P}$ | Field or Method is in a type (method/class)<br>$\langle md, T\rangle \in_{P} c \Leftrightarrow \exists T, V s.t. \langle md, T, V, e\rangle \in^{\mathsf{c}}_{P} c$ |
| $\in_{P}$ | Field or Method is in a type (field/type)<br>$\langle c.fd, t\rangle \in_{P} c \Leftrightarrow \langle c.fd, t\rangle \in^{\mathsf{c}}_{P} c$ |
| $\mathsf{Pre}_{P}$ | Pre-condition contract is in method of interface<br>$e\ \mathsf{Pre}_{P}\ \langle i, md\rangle \Leftrightarrow$ **interface** $i \{\cdots t\ md\ arg$ **@pre** $\{ e \}$ **@post** $\{ e' \} \cdots\}$ is in $P$ |
| $\mathsf{Pre}_{P}$ | Pre-condition contract is in method of class<br>$e\ \mathsf{Pre}_{P}\ \langle c, md\rangle \Leftrightarrow$ **class** $c \{\cdots t\ md\ arg \{ body \}$ **@pre** $\{ e \}$ **@post** $\{ e' \} \cdots\}$ is in $P$ |
| $\mathsf{Post}_{P}$ | Post-condition contract is in method of interface<br>$e\ \mathsf{Post}_{P}\ \langle i, md\rangle \Leftrightarrow$ **interface** $i \{\cdots t\ md\ arg$ **@pre** $\{ e' \}$ **@post** $\{ e \} \cdots\}$ is in $P$ |
| $\mathsf{Post}_{P}$ | Post-condition contract is in method of class<br>$e\ \mathsf{Post}_{P}\ \langle c, md\rangle \Leftrightarrow$ **class** $c \{\cdots t\ md\ arg \{ body \}$ **@pre** $\{ e' \}$ **@post** $\{ e \} \cdots\}$ is in $P$ |

Figure 4.3: Predicates and relations in the model of Contract Java, ii

indicating that the method must be overridden in a subclass before the class is instantiated. Unlike Java, the body of a method is just an expression whose result is the result of the method. Like Java, classes are instantiated with the **new** operator, but there are no class constructors[†] in Contract Java; instance variables are initialized to **null**. Finally, the **view** and **let** forms represent Java's casting expressions and the capability for binding variables locally. In the code examples presented in this paper, we omit the **extends** and **implements** clauses when nothing would appear after them.

The type checker translates syntax (a) to syntax (b). It inserts additional information (underlined in figure 4.1 (b)) to be used by the contract elaborator and the evaluator. To support contract elaboration, method calls are annotated with the type of the object whose method is called. To support evaluation, field update and field reference are annotated with the class containing the field, and calls to **super** are annotated with the class.

The contract elaborator produces syntax (c) and the evaluator accepts it. The @**pre** and @**post** conditions are removed from interfaces and classes, and the contract expressions are inserted elsewhere in the elaborated program. Syntax (c) also adds three constructs to the language: **preErr**, **postErr**, and **hierErr**. These constructs are used to signal contract violations.

Expressions of the shape:

$$\textbf{return} : t,\ c\ \{\ e\ \}$$

mark method returns. The type $t$ indicates the type of the object whose method was invoked, in parallel to the type annotations on method calls, and the class name, $c$, is the class that defined the invoked method. Unlike standard Java, in Contract Java the programmer does not write **return** expressions in the program. Instead, the evaluator introduces **return** expressions as it executes the program. They are annotations that are used in the statement and the proof of the contract soundness theorem.

[†]Pre- and post-condition contracts for constructors can be treated as contracts on methods that are never overridden.

A valid Contract Java program satisfies a number of predicates; these are described in Figures 4.2 and 4.3. For example, the CLASSESONCE($P$) predicate states that each class name is defined at most once in the program $P$. Additionally, there are a number of relations on the syntax of a valid Contract Java program. The relation $\prec^c_P$ associates each class name in $P$ to the class it extends, and the (overloaded) $\in^c_P$ relations capture the field and method declarations of $P$.

The syntax-summarizing relations induce a second set of relations and predicates that summarize the class structure of a program. The first of these is the subclass relation $\leq^c_P$, which is a partial order if the COMPLETECLASSES($P$) and WELLFOUNDEDCLASSES($P$) predicates hold. In this case, the classes declared in $P$ form a tree that has **Object** at its root.

If the program describes a tree of classes, we can decorate each class in the tree with the collection of fields and methods that it accumulates from local declarations and inheritance. The source declaration of any field or method in a class can be computed by finding the *minimum* superclass (*i.e.*, farthest from the root) that declares the field or method. This algorithm is described precisely by the $\in^c_P$ relations. The $\in^c_P$ relation retains information about the source class of each field, but it does not retain the source class for a method. This reflects the property of Java classes that fields cannot be overridden (so instances of a subclass always contain the field), while methods can be overridden (and may become inaccessible).

Interfaces have a similar set of relations. The superinterface declaration relation $\prec^i_P$ induces a subinterface relation $\leq^i_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The set methods of an interface, as described by $\in^i_P$, is the union of the interface's declared methods and the methods of its superinterfaces.

Classes and interfaces are related by **implements** declarations, as captured in the $\prec\!\!\prec^c_P$ relation. This relation is a set of edges joining the class tree and the interface graph, completing the subtype picture of a program. A type in the full graph is a subtype of all of its ancestors.

The subtype structure of a program is captured by the $\leq_P$ relation.A type $t$ is a subtype of another type $t'$ in a program $P$, written $t \leq_P t'$, when one of these conditions holds:

- $t$ and $t'$ are the same type,
- $t$ and $t'$ are both classes, $t$ is derived from $t''$ in $P$, and $t'' \leq_P t'$,
- $t$ and $t'$ are both interfaces, $t$ is an extension of $t''$ in $P$ (also written $t \prec^{\mathsf{i}}_P t''$), and $t'' \leq_P t'$, or
- $t$ is a class and $t'$ is an interface, and either
  - $t$ implements $t'$ in $P$,
  - $t$ implements an interface $i$ in $P$ and $i \leq_P t'$ , or
  - $t$ is derived from a class $c$ in $P$ and $c \leq_P t'$.

The relations $\mathsf{PRE}_P$ and $\mathsf{POST}_P$ relate expressions with pairs of methods and types. An expression $e$ is the pre-condition for $m$ in $t$ in the program $P$, written $e\ \mathsf{PRE}_P\ \langle t, m\rangle$, if the expression $e$ appears in the program $P$, declared as a precondition of $m$ in $t$. Similarly an expression $e$ is a postcondition of $m$ in $t$ in the program $P$ if $e\ \mathsf{POST}_P\ \langle t, m\rangle$.

## 4.3 Type Elaboration

The type elaboration rules for Contract Java are defined by the following judgements:

| | |
|---|---|
| $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ | $P$ elaborates to $P'$ with type $t$ |
| $P \vdash_{\mathsf{d}} \mathit{defn} \Rightarrow \mathit{defn}'$ | *defn* elaborates to *defn*′ |
| $P, c \vdash_{\mathsf{m}} \mathit{meth} \Rightarrow \mathit{meth}'$ | *meth* in class $c$ elaborates to *meth*′ |
| $P, i \vdash_{\mathsf{i}} \mathit{imeth} \Rightarrow \mathit{imeth}'$ | *imeth* in interface $i$ elaborates to *imeth*′ |
| $P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$ | $e$ elaborates to $e'$ with type $t$ in $\Gamma$ |
| $P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$ | $e$ has type $t$ using subsumption in $\Gamma$ |
| $P \vdash_{\mathsf{t}} t$ | $t$ is a valid type |

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{l}\textsc{ClassesOnce}(P)\ \ \textsc{InterfacesOnce}(P)\ \ \textsc{MethodOncePerClass}(P)\ \ \textsc{FieldOncePerClass}(P)\ \ \textsc{CompleteClasses}(P)\\ \textsc{WellFoundedClasses}(P)\ \ \textsc{CompleteInterfaces}(P)\ \ \textsc{WellFoundedInterfaces}(P)\ \ \textsc{InterfaceMethodsOK}(P)\\ \textsc{MethodArgsDistinct}(P)\ \ \textsc{ClassesImplementAll}(P)\ \ P \vdash_{\mathsf{d}} \mathit{defn}_j \Rightarrow \mathit{defn}'_j \text{ for } j \in [1,n]\ \ P,[\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t\\ \text{where } P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e\end{array}}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ldots \mathit{defn}_n\ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e' : t}$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} \mathit{meth}_k \Rightarrow \mathit{meth}'_k \text{ for } k \in [1,p]}{\begin{array}{l}P \vdash_{\mathsf{d}} \textbf{class}\ c\ \cdots\ \{\ t_1\ \mathit{fd}_1\ \ldots\ t_n\ \mathit{fd}_n\ \ \mathit{meth}_1\ \ldots\ \mathit{meth}_p\ \}\\ \quad \Rightarrow \textbf{class}\ c\ \cdots\ \{\ t_1\ \mathit{fd}_1\ \ldots\ t_n\ \mathit{fd}_n\ \ \mathit{meth}'_1\ \ldots\ \mathit{meth}'_p\ \}\end{array}}$$

$$\frac{P \vdash_{\mathsf{i}} \mathit{imeth}_j \Rightarrow \mathit{imeth}'_j \text{ for } j \in [1,p]}{P, i \vdash_{\mathsf{d}} \textbf{interface}\ i\ \cdots\ \{\ \mathit{imeth}_1\ \ldots\ \mathit{imeth}_p\ \} \Rightarrow \textbf{interface}\ i\ \cdots\ \{\ \mathit{imeth}'_1\ \ldots\ \mathit{imeth}'_p\ \}}$$

$\vdash_{\mathsf{m}}$

$$\frac{\begin{array}{l}P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P,[\mathit{this} : t_o, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{s}} e \Rightarrow e' : t\\ P,[\mathit{this} : t_o, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{e}} e_b \Rightarrow e'_b : \textbf{boolean}\ \ P,[\mathit{this} : t_o, \textbf{@ret} : t] \vdash_{\mathsf{e}} e_a \Rightarrow e'_a : \textbf{boolean}\end{array}}{\begin{array}{l}P, t_o \vdash_{\mathsf{m}} t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ e\ \}\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \}\\ \quad \Rightarrow t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ e'\ \}\ \textbf{@pre}\ \{\ e'_b\ \}\ \textbf{@post}\ \{\ e'_a\ \}\end{array}}$$

$$\frac{\begin{array}{l}P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P,[\mathit{this} : t_o, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{e}} e_b \Rightarrow e'_b : \textbf{boolean}\\ P,[\mathit{this} : t_o, \textbf{@ret} : t] \vdash_{\mathsf{e}} e_a \Rightarrow e'_a : \textbf{boolean}\end{array}}{\begin{array}{l}P, t_o \vdash_{\mathsf{m}} t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ \textbf{abstract}\ \}\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \}\\ \quad \Rightarrow t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ \textbf{abstract}\ \}\ \textbf{@pre}\ \{\ e'_b\ \}\ \textbf{@post}\ \{\ e'_a\ \}\end{array}}$$

$\vdash_{\mathsf{i}}$

$$\frac{\begin{array}{l}P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P,[\mathit{this} : i, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{e}} e_b \Rightarrow e'_b : \textbf{boolean}\\ P,[\mathit{this} : i, \textbf{@ret} : t] \vdash_{\mathsf{e}} e_a \Rightarrow e'_a : \textbf{boolean}\end{array}}{P, i \vdash_{\mathsf{i}} t\ \mathit{md}\ (t_1\ \mathit{arg}_1\ \ldots\ t_n\ \mathit{arg}_n)\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \Rightarrow t\ \mathit{md}\ (t_1\ \mathit{arg}_1\ \ldots\ t_n\ \mathit{arg}_n)\ \textbf{@pre}\ \{\ e'_b\ \}\ \textbf{@post}\ \{\ e'_a\ \}}$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} c \qquad \textsc{NoAbstractMethods}(P, c)}{P, \Gamma \vdash_{\mathsf{e}} \textbf{new}\ c \Rightarrow \textbf{new}\ c : c} \qquad \frac{\mathit{var} \in \mathrm{dom}(\Gamma)}{P, \Gamma \vdash_{\mathsf{e}} \mathit{var} \Rightarrow \mathit{var} : \Gamma(\mathit{var})}$$

$$\frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{null} \Rightarrow \textbf{null} : t} \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle c.\mathit{fd}, t\rangle \in_P t'}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} \Rightarrow e'\underline{\ : c\ }.\mathit{fd} : t}$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle c.\mathit{fd}, t\rangle \in_P t' \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} = e_v \Rightarrow e'\underline{\ : c\ }.\mathit{fd} = e'_v : t}$$

Figure 4.4: Context-sensitive Checks and Type Elaboration Rules for Contract Java, i

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle md, (t_1 \ldots t_n \longrightarrow t)\rangle \in_P t' \qquad P,\Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n]}{P,\Gamma \vdash_{\mathsf{e}} e.md\,(e_1 \ldots e_n) \Rightarrow e' \underline{: t'}.md\,(e'_1 \ldots e'_n) : t}$$

$$\frac{\begin{array}{l} P,\Gamma \vdash_{\mathsf{e}} \mathit{this} \Rightarrow \mathit{this} : c' \qquad c' \prec^{c}_{P} c \qquad \langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e_b\rangle \in^{c}_{P} c \\ P,\Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \qquad e_b \neq \textbf{abstract} \end{array}}{P,\Gamma \vdash_{\mathsf{e}} \textbf{super}.md(e_1 \ldots e_n) \Rightarrow \textbf{super}\underline{\equiv \mathit{this} : c}.md(e'_1 \ldots e'_n) : t}$$

$$\frac{P,\Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P,\Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow e' : t}$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t \leq_P t' \text{ or } t \in \mathrm{dom}(\prec^{i}_{P}) \text{ or } t' \in \mathrm{dom}(\prec^{i}_{P})}{P,\Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow \textbf{view}\ t\ e' : t}$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \qquad P,\Gamma[var_1 : t_1]\cdots[var_n : t_n] \vdash_{\mathsf{e}} e \Rightarrow e' : t}{P,\Gamma \vdash_{\mathsf{e}} \textbf{let}\ \{\ var_1 = e_1 \cdots var_n = e_n\ \}\ \textbf{in}\ e \Rightarrow \textbf{let}\ \{\ var_1 = e'_1 \cdots var_n = e'_n\ \}\ \textbf{in}\ e' : t}$$

$$\frac{}{P,\Gamma \vdash_{\mathsf{e}} \textbf{true} \Rightarrow \textbf{true} : \textbf{boolean}} \qquad \frac{}{P,\Gamma \vdash_{\mathsf{e}} \textbf{false} \Rightarrow \textbf{false} : \textbf{boolean}}$$

$$\frac{\begin{array}{l} P,\Gamma \vdash_{\mathsf{e}} e_1 \Rightarrow e'_1 : \textbf{boolean} \qquad P,\Gamma \vdash_{\mathsf{e}} e_2 \Rightarrow e'_2 : t \\ P,\Gamma \vdash_{\mathsf{e}} e_3 \Rightarrow e'_3 : t \end{array}}{P,\Gamma \vdash_{\mathsf{e}} \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 \Rightarrow \textbf{if}\ (e'_1)\ e'_2\ \textbf{else}\ e'_3 : t} \qquad \frac{P,\Gamma \vdash_{\mathsf{e}} e_1 \Rightarrow e'_1 : t' \qquad P,\Gamma \vdash_{\mathsf{e}} e_2 \Rightarrow e'_2 : t}{P,\Gamma \vdash_{\mathsf{e}} \{\ e_1 ; e_2\ \} \Rightarrow \{\ e_1 ; e_2\ \} : t}$$

$\vdash_{\mathsf{s}}, \vdash_{\mathsf{t}}$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \leq_P t}{P,\Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t} \qquad \frac{t \in \mathrm{dom}(\prec^{c}_{P}) \cup \mathrm{dom}(\prec^{i}_{P}) \cup \{\textbf{Object}, \mathsf{Empty}, \textbf{boolean}\}}{P \vdash_{\mathsf{t}} t}$$

Figure 4.5: Context-sensitive Checks and Type Elaboration Rules for Contract Java, ii

The complete typing rules are shown in figures 4.4 and 4.5. A program is well-typed if its class definitions and final expression are well-typed. A definition, in turn, is well-typed when its field and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types. For example, the **get**$^{c}$ and **set**$^{c}$ rules for fields first determine the type of the instance expression, and then calculate a class-tagged field name using $\in_P$; this yields both the type of the field and the class for the installed annotation. In the **set**$^{c}$ rule, the right-hand side of the assignment must match the type of the field, but this match may exploit subsumption to coerce the type of the value to a supertype.

The type elaboration rules translate expressions that access a field, call a **super** method, or call a normal method into annotated expressions (see the underlined parts of Figure 4.1). For field uses, the annotated expression contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. For **super** method invocations, the annotated expression contains the compile-time type of *this*, which determines the class that contains the declaration of the method to be invoked. For regular method calls, the annotation contains the type of the object being called.

The contract expressions are well-typed if they have type **boolean**. In addition, pre-condition expressions may contain references to the arguments of the checked method and *this*, and post-condition expressions may contain references to the result of the method, and *this*. The variable **@ret** refers to the result of the method in the post-condition.

## 4.4 Contract Elaboration

Contract checking is modeled as a translation, called *Elab*, from syntax (b) to syntax (c). Since contract checking is triggered via method calls, we need to understand how *Elab* must deal with those. Consider the following code fragment:

$$\begin{array}{l} \mathit{IConsole}\ o\ =\ \mathit{Factory}.\mathit{newConsole}(\cdots); \\ \cdots\ o.\mathit{display}(\text{“It's crunch time.”})\ \cdots \end{array}$$

Since the programmer cannot know what kind of console *o* represents at run-time, he can establish only the preconditions for *display* that *IConsole* specifies. Hence, the code that *Elab* produces for the method call must first test the preconditions for *display* in *IConsole*. If this test fails, the author of the method call has made a mistake. If the test succeeds, the contract monitoring code can check the ancestor portion of the class and interface hierarchy that is determined by *o*'s class tag.[‡] These hierarchy checks ensure that the precondition of an overridden method implies the precondition of the overriding method, and that the postcondition of an overriding method implies the postcondition of each overridden method.

To perform both forms of checking, *Elab* adds new classes to check the subtype hierarchy and inserts methods into existing classes to check pre- and post-conditions. For each method of a class, the elaborator inserts several wrapper methods, one for each type that instances of the class might have. These wrapper methods perform the pre- and post-condition checking and call the hierarchy checkers. Additionally, the elaborator redirects each method call so it invokes the appropriate wrapper method, based on the static type of the object whose method is invoked.

The translation given in this chapter differs from the one in the previous chapter. In the previous chapter, hierarchy checking methods were added to each class. Here, the hierarchy checking methods are separated into their own classes. This simplifies the model and the proof of contract soundness. Both translations check the same contracts for any given program.

In the above invocation, the elaborator inserts a *display_IConsole* wrapper method into the each console class since each console class can be cast to *IConsole*. Additionally, it rewrites the call to the *display* method to call the *display_IConsole* method, since *o*'s type is *IConsole*. Each *display_IConsole* method checks *IConsole*'s pre-condition and the pre-condition hierarchy from the instantiated class upwards. Then the *display_Console*

[‡]Following ML tradition, we use the word "type" to refer only to the static type determined by the type checker. We use the words "class tag" to refer to the so-called dynamic or run-time type.

**prog**

$$\frac{P \vdash defn_j \rightharpoonup_{\mathsf{d}} defn'_j\ defn_{j_{pre}}\ defn_{j_{post}}\ \text{for } j \in [1,n] \qquad P, main \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad \text{where } P = defn_1 \ldots defn_n\ e}{\vdash defn_1 \ldots defn_n\ e \rightharpoonup_{\mathsf{p}} defn'_1\ defn_{1_{pre}}\ defn_{1_{post}} \ldots defn'_n\ defn_{n_{pre}}\ defn_{n_{post}}\ e'}$$

**defn$^{\mathsf{i}}$**

$$\frac{\vdash imeth_j \rightharpoonup_{\mathsf{i}} imeth'_j \qquad P, c \vdash imeth_j \rightharpoonup_{\mathsf{pre}} meth_{j_{pre}} \qquad P, c \vdash imeth_j \rightharpoonup_{\mathsf{post}} meth_{j_{post}} \qquad \text{for } j \in [1,n]}{\begin{array}{l} P \vdash \textbf{interface}\ i\ \textbf{extends}\ i_1 \ldots i_l\ imeth_1 \ldots imeth_n \rightharpoonup_{\mathsf{d}} \textbf{interface}\ i\ \textbf{extends}\ i_1 \ldots i_l\ imeth'_1 \ldots imeth'_n \\ \qquad \textbf{class}\ check\_i\_pre\ imeth_{1_{pre}} \cdots imeth_{n_{pre}} \\ \qquad \textbf{class}\ check\_i\_post\ imeth_{1_{post}} \cdots imeth_{n_{post}} \end{array}}$$

**defn$^{\mathsf{c}}$**

$$\frac{\begin{array}{l} P, c \vdash meth_j \rightharpoonup_{\mathsf{m}} meth'_j \qquad P, c \vdash meth_j \rightharpoonup_{\mathsf{pre}} meth_{j_{pre}} \qquad P, c \vdash meth_j \rightharpoonup_{\mathsf{post}} meth_{j_{post}} \qquad \text{for } j \in [1,n] \\ P, c, t \vdash meth_j \rightharpoonup_{\mathsf{w}} wrap\_method_j \qquad \text{for } j \in [1,n], \text{ and } t \text{ such that } c \leq_P t \end{array}}{\begin{array}{l} P \vdash \textbf{class}\ c\ \textbf{extends}\ c'\ \textbf{implements}\ i_1 \ldots i_l \rightharpoonup_{\mathsf{d}} \textbf{class}\ c\ \textbf{extends}\ c'\ \textbf{implements}\ i_1 \ldots i_l \\ \qquad meth_1 \ldots meth_n \qquad\qquad meth'_1 \ldots meth'_n \\ \qquad\qquad wrap\_method_1 \ldots wrap\_method_n \ldots \\ \qquad\qquad \textbf{class}\ check\_c\_pre\ \textbf{extends Object}\ meth_{1_{pre}} \cdots meth_{n_{pre}} \\ \qquad\qquad \textbf{class}\ check\_c\_post\ \textbf{extends Object}\ meth_{1_{post}} \cdots meth_{n_{post}} \end{array}}$$

**meth$^{\mathsf{i}}$**

$$\frac{}{P \vdash t\ md\,(t_1\ var_1 \ldots t_n\ var_n)\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \rightharpoonup_{\mathsf{i}} t\ md\,(t_1\ var_1 \ldots t_n\ var_n)}$$

**meth$^{\mathsf{c}}$**

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash t\ md\,(\ t_1\ arg_1 \ldots t_n\ arg_n\ )\ \{\ e\ \}\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \rightharpoonup_{\mathsf{m}} t\ md\,(\ t_1\ arg_1 \ldots t_n\ arg_n\ )\ \{\ e'\ \}}$$

**wrap**

$$\frac{e_b\ \textsc{Pre}_P\ \langle t, md\rangle \qquad e_a\ \textsc{Post}_P\ \langle t, md\rangle \qquad P, c \vdash e_b \rightharpoonup_{\mathsf{e}} e'_b \qquad P, c \vdash e_a \rightharpoonup_{\mathsf{e}} e'_a}{P, c, t \vdash t'\ md\,(t_1\ x_1, \cdots, t_j\ x_j) \cdots \rightharpoonup_{\mathsf{w}}}$$

```
t' _md (t1 x1, ···, tj xj, string cname) {
  if (e'_b) {
      (new check_c_pre()).md(this, x1, ···, xj);
      let { @ret = this.md(x1, ···, xj) }
      in { if (e'_a)
             (new check_c_post()).md("dummy", true, this, md, x1, ···, xj);
           else
             postErr(c);
           md }
  } else {
    preErr(cname);
  }
}
```

Figure 4.6: Blame Compilation, i

method calls the original *display* method. When it returns, the *display_IConsole* method checks *IConsole*'s post-condition and the post-condition hierarchy from the instantiated class upwards. The rest of this section presents the elaborator both concretely via the example of the console classes and interfaces, and abstractly via judgements that define the elaborator.

**prehier**$^{\mathsf{c}}$

$$
\frac{c \prec^{\mathsf{c}}_{P} \textbf{Object} \qquad \text{for all } i \text{ such that } c \ll^{\mathsf{c}}_{P} i}
{\begin{array}{l}
P, c \vdash t \;\; md \; (t_1 \;\; var_1 \; \ldots \; t_n \;\; var_n) \; \{ \; e \; \} \\
\qquad \textbf{@pre} \; \{ \; e_b \; \} \\
\qquad \textbf{@post} \; \{ \; e_a \; \}
\end{array}
\rightharpoonup_{\mathsf{pre}}
\begin{array}{l}
\textbf{boolean} \;\; md \; (c \;\; this, \; t_1 \;\; var_1, \; \cdots, \; t_n \;\; var_n) \; \{ \\
\quad \textbf{let} \; \{ \; next = (\textbf{new} \;\; check\_i\_pre()).md(this, \; var_1, \; \cdots, \; var_n) \; || \; \cdots \\
\qquad\quad res = e_a \; \} \\
\quad \textbf{in} \;\; \textbf{if} \; (!next \; || \; res) \; /\!/ \; next \Rightarrow res \\
\qquad\quad res \\
\qquad \textbf{else} \\
\qquad\quad \textbf{hierErr}(c) \\
\}
\end{array}}
$$

$$
\frac{c \prec^{\mathsf{c}}_{P} c' \qquad c' \neq \textbf{Object} \qquad \text{for all } i \text{ such that } c \ll^{\mathsf{c}}_{P} i}
{\begin{array}{l}
P, c \vdash t \;\; md \; (t_1 \;\; var_1 \; \ldots \; t_n \;\; var_n) \; \{ \; e \; \} \\
\qquad \textbf{@pre} \; \{ \; e_b \; \} \\
\qquad \textbf{@post} \; \{ \; e_a \; \}
\end{array}
\rightharpoonup_{\mathsf{pre}}
\begin{array}{l}
\textbf{boolean} \;\; md \; (c \;\; this, \; t_1 \;\; var_1, \; \cdots, \; t_n \;\; var_n) \; \{ \\
\quad \textbf{let} \; \{ \; next = (\textbf{new} \;\; check\_c'\_pre()).md(this, \; var_1, \; \cdots, \; var_n) \; || \\
\qquad\qquad\qquad (\textbf{new} \;\; check\_i\_pre()).md(this, \; var_1, \; \cdots, \; var_n) \; || \; \cdots \\
\qquad\quad res = e_a \; \} \\
\quad \textbf{in} \;\; \textbf{if} \; (!next \; || \; res) \; /\!/ \; next \Rightarrow res \\
\qquad\quad res \\
\qquad \textbf{else} \\
\qquad\quad \textbf{hierErr}(c) \\
\}
\end{array}}
$$

**prehier**$^{\mathsf{i}}$

$$
\frac{\text{for all } i' \text{ such that } i \prec^{\mathsf{i}}_{P} i'}
{\begin{array}{l}
P, i \vdash t \;\; md \; (t_1 \;\; var_1 \; \ldots \; t_n \;\; var_n) \; \{ \; e \; \} \\
\qquad \textbf{@pre} \; \{ \; e_b \; \} \\
\qquad \textbf{@post} \; \{ \; e_a \; \}
\end{array}
\rightharpoonup_{\mathsf{pre}}
\begin{array}{l}
\textbf{boolean} \;\; md \; (i \;\; this, \; t_1 \;\; var_1, \; \ldots \; t_n \;\; var_n) \; \{ \\
\quad \textbf{let} \; \{ \; next = (\textbf{new} \;\; check\_i'\_pre()).md(this, \; var_1, \; \cdots, \; var_n) \; || \; \cdots \\
\qquad\quad res = e_b \; \} \\
\quad \textbf{in} \;\; \textbf{if} \; (!next \; || \; res) \; /\!/ \; next \Rightarrow res \\
\qquad\quad res \\
\qquad \textbf{else} \\
\qquad\quad \textbf{hierErr}(i) \\
\}
\end{array}}
$$

Figure 4.7: Blame Compilation, ii

$\boxed{\textbf{posthier}^{\mathsf{c}}}$

$$
\frac{c \prec^{c}_{\mathcal{P}} \textbf{Object} \qquad \text{for all } i \text{ such that } c \ll^{c}_{\mathcal{P}} i}
{\begin{array}{lll}
P, c \vdash t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ e\ \} & \rightharpoonup_{\mathsf{post}} & \textbf{boolean}\ \mathit{md}\ (\textbf{String}\ \mathit{tbb},\ \textbf{boolean}\ \mathit{last},\ c\ \mathit{this},\ t\ \mathit{md},\ t_1\ \mathit{var}_1,\ \cdots,\ t_n\ \mathit{var}_n)\ \{ \\
\quad \textbf{@pre}\ \{\ e_b\ \} & & \quad \textbf{let}\ \{\ \mathit{res} = e_a\ \} \\
\quad \textbf{@post}\ \{\ e_a\ \} & & \quad \textbf{in if}\ (!\mathit{last}\ ||\ \mathit{res})\ //\ \mathit{last} \Rightarrow \mathit{res} \\
 & & \qquad\quad (\textbf{new}\ \mathit{check\_i\_post}()).\mathit{md}(c,\ \mathit{res},\ \mathit{this},\ \mathit{md},\ \mathit{var}_1,\ \cdots,\ \mathit{var}_n)\ \&\&\ \cdots \\
 & & \qquad \textbf{else} \\
 & & \qquad\quad \textbf{hierErr}(\mathit{tbb}) \\
 & & \}
\end{array}}
$$

$$
\frac{c \prec^{c}_{\mathcal{P}} c' \qquad c' \neq \textbf{Object} \qquad \text{for all } i \text{ such that } c \ll^{c}_{\mathcal{P}} i}
{\begin{array}{lll}
P, c \vdash t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ e\ \} & \rightharpoonup_{\mathsf{post}} & \textbf{boolean}\ \mathit{md}\ (\textbf{String}\ \mathit{tbb},\ \textbf{boolean}\ \mathit{last},\ c\ \mathit{this},\ t\ \mathit{md},\ t_1\ \mathit{var}_1,\ \cdots,\ t_n\ \mathit{var}_n)\ \{ \\
\quad \textbf{@pre}\ \{\ e_b\ \} & & \quad \textbf{let}\ \{\ \mathit{res} = e_a\ \} \\
\quad \textbf{@post}\ \{\ e_a\ \} & & \quad \textbf{in if}\ (!\mathit{last}\ ||\ \mathit{res})\ //\ \mathit{last} \Rightarrow \mathit{res} \\
 & & \qquad\quad (\textbf{new}\ \mathit{check\_c'\_post}()).\mathit{md}(c,\ \mathit{res},\ \mathit{this},\ t\ \mathit{md},\ \mathit{var}_1,\ \cdots,\ \mathit{var}_n)\ \&\& \\
 & & \qquad\quad (\textbf{new}\ \mathit{check\_i\_post}()).\mathit{md}(c,\ \mathit{res},\ \mathit{this},\ t\ \mathit{md},\ \mathit{var}_1,\ \cdots,\ \mathit{var}_n)\ \&\&\ \cdots \\
 & & \qquad \textbf{else} \\
 & & \qquad\quad \textbf{hierErr}(\mathit{tbb}) \\
 & & \}
\end{array}}
$$

$\boxed{\textbf{posthier}^{\mathsf{i}}}$

$$
\frac{\text{for all } i' \text{ such that } i \prec^{i}_{\mathcal{P}} i'}
{\begin{array}{lll}
P, i \vdash t\ \mathit{md}\ (t_1\ \mathit{var}_1\ \ldots\ t_n\ \mathit{var}_n)\ \{\ e\ \} & \rightharpoonup_{\mathsf{post}} & \textbf{boolean}\ \mathit{md}\ (\textbf{String}\ \mathit{tbb},\ \textbf{boolean}\ \mathit{last},\ i\ \mathit{this},\ t\ \mathit{md},\ t_1\ \mathit{var}_1,\ \cdots,\ t_n\ \mathit{var}_n)\ \{ \\
\quad \textbf{@pre}\ \{\ e_b\ \} & & \quad \textbf{let}\ \{\ \mathit{res} = e_a\ \} \\
\quad \textbf{@post}\ \{\ e_a\ \} & & \quad \textbf{in if}\ (!\mathit{last}\ ||\ \mathit{res})\ //\ \mathit{last} \Rightarrow \mathit{res} \\
 & & \qquad\quad (\textbf{new}\ \mathit{check\_i'\_post}()).\mathit{md}(c,\ \mathit{res},\ \mathit{this},\ \mathit{md},\ \mathit{var}_1,\ \cdots,\ \mathit{var}_n)\ \&\&\ \cdots \\
 & & \qquad \textbf{else} \\
 & & \qquad\quad \textbf{hierErr}(\mathit{tbb}) \\
 & & \}
\end{array}}
$$

Figure 4.8: Blame Compilation, iii

$\boxed{\textbf{exp}}$

$$\frac{}{P, c \vdash \textbf{new}\ c' \rightharpoonup_{\mathsf{e}} \textbf{new}\ c'} \qquad \frac{}{P, c \vdash \textbf{null} \rightharpoonup_{\mathsf{e}} \textbf{null}} \qquad \frac{}{P, c \vdash \mathit{var} \rightharpoonup_{\mathsf{e}} \mathit{var}}$$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c \vdash e_v \rightharpoonup_{\mathsf{e}} e'_v}{P, c \vdash e \underline{\ :\ c'\ }.\mathit{fd} = e_v \rightharpoonup_{\mathsf{e}} e' \underline{\ :\ c'\ }.\mathit{fd} = e'_v} \qquad \frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash e \underline{\ :\ c'\ }.\mathit{fd} \rightharpoonup_{\mathsf{e}} e' \underline{\ :\ c'\ }.\mathit{fd}}$$

$\boxed{\textbf{call}^{\mathsf{i}}}$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c \vdash e_j \underline{:\ i} \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1, n]}{P, c \vdash e \underline{:\ i}.\mathit{md}\,(e_1, \cdots, e_n) \rightharpoonup_{\mathsf{e}} e'.i\_\mathit{md}(e'_1, \cdots, e'_n)}$$

$$\frac{P, c' \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c' \vdash e_j \underline{:\ c} \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1, n]}{P, c' \vdash e \underline{:\ c}.\mathit{md}\,(e_1, \cdots, e_n) \rightharpoonup_{\mathsf{e}} e.c\_\mathit{md}\,(c', e_1 \cdots, e_n\ t)}$$

$$\frac{P, c \vdash e_j \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1, n]}{P, c \vdash \textbf{super} \underline{\equiv \mathit{this} : c'\ }.\mathit{md}(e_1 \ldots e_n) \rightharpoonup_{\mathsf{e}} \textbf{super} \underline{\equiv \mathit{this} : c'\ }.c'\_\mathit{md}(e'_1 \ldots e'_n)}$$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash \textbf{view}\ t\ e \rightharpoonup_{\mathsf{e}} \textbf{view}\ t\ e'} \qquad \frac{P, c \vdash e_j \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1, n] \qquad P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash \textbf{let}\ \{\ \mathit{var}_1 = e_1 \cdots \mathit{var}_n = e_n\ \}\ \textbf{in}\ e \rightharpoonup_{\mathsf{e}} \textbf{let}\ \{\ \mathit{var}_1 = e'_1 \cdots \mathit{var}_n = e'_n\ \}\ \textbf{in}\ e'}$$

$$\frac{}{P, c \vdash \textbf{true} \rightharpoonup_{\mathsf{e}} \textbf{true}}$$

$$\frac{}{P, c \vdash \textbf{false} \rightharpoonup_{\mathsf{e}} \textbf{false}}$$

$$\frac{P, c \vdash e_1 \rightharpoonup_{\mathsf{e}} e'_1 \qquad P, c \vdash e_2 \rightharpoonup_{\mathsf{e}} e'_2 \qquad P, c \vdash e_3 \rightharpoonup_{\mathsf{e}} e'_3}{P, c \vdash \textbf{if}\,(e_1)\ e_2\ \textbf{else}\ e_3 \rightharpoonup_{\mathsf{e}} \textbf{if}\,(e'_1)\ e'_2\ \textbf{else}\ e'_3} \qquad \frac{P, c \vdash e_1 \rightharpoonup_{\mathsf{e}} e'_1 \qquad P, c \vdash e_2 \rightharpoonup_{\mathsf{e}} e'_2}{P, c \vdash \{\ e_1\ ;\ e_2\ \} \rightharpoonup_{\mathsf{e}} \{\ e'_1\ ;\ e'_2\ \}}$$

Figure 4.9: Blame Compilation, iv

Formally, our contract elaborator is defined by these judgements:

$\vdash P \rightharpoonup_{\mathsf{p}} P'$
The program $P$ compiles to the program $P'$.

$P \vdash \mathit{defn} \rightharpoonup_{\mathsf{d}} \mathit{defn}'\ \mathit{defn}_{pre}\ \mathit{defn}_{post}$
*defn* compiles to $\mathit{defn}'$ with checkers $\mathit{defn}_{pre}$ and $\mathit{defn}_{post}$ in $P$.

$P \vdash \mathit{imeth} \rightharpoonup_{\mathsf{i}} \mathit{imeth}'$
*imeth* compiles to $\mathit{imeth}'$.

$P, c \vdash \mathit{meth} \rightharpoonup_{\mathsf{m}} \mathit{meth}'$
*meth* compiles to $\mathit{meth}'$ in class $c$.

$P, c, t \vdash \mathit{meth} \rightharpoonup_{\mathsf{w}} \mathit{meth}'$
$\mathit{meth}'$ checks the pre- and post-conditions for $t$'s *meth*,
which blames $c$ for contract violations.

$P, c \vdash e \rightharpoonup_{\mathsf{e}} e'$
$e$ compiles to $e'$, which blames $c$ for contract violations.

$P, t \vdash \mathit{imeth} \rightharpoonup_{\mathsf{pre}} \mathit{imeth}'$
$\mathit{imeth}'$ checks the hierarchy for the pre-condition of *imeth* in $t$.

$P, t \vdash \mathit{imeth} \rightharpoonup_{\mathsf{post}} \mathit{imeth}'$
$\mathit{imeth}'$ checks the hierarchy for the post-condition of *imeth* in $t$.

The first judgement, $\rightharpoonup_{\mathsf{p}}$, is the program elaboration judgement. It defines *Elab*:

$$\mathit{Elab}(P) = P' \text{ where } P \rightharpoonup_{\mathsf{p}} P'$$

Occasionally, *Elab* is also applied to other syntactic categories, such as expressions. In that case, the context dictates an implied program that *Elab* is applied to and the result is the elaborated expression, from that program.

The $\rightharpoonup_{\mathsf{d}}$ judgement builds three definitions for each definition in the original program. The first is derived from the original definition. The second and third are the pre- and post-condition hierarchy checking classes, respectively.

The $\rightharpoonup_{\mathsf{i}}$ judgement removes interface method contracts. The $\rightharpoonup_{\mathsf{m}}$, $\rightharpoonup_{\mathsf{e}}$, and $\rightharpoonup_{\mathsf{w}}$ judgements produce the annotated class. The $\rightharpoonup_{\mathsf{w}}$ judgement constructs the wrapper methods that check the contracts. The $\rightharpoonup_{\mathsf{m}}$ judgement re-writes methods and removes class method contracts. The $\rightharpoonup_{\mathsf{e}}$ judgement rewrites expressions so that method calls are re-directed to the wrapper methods, based on the type of the call. The final two judgements, $\rightharpoonup_{\mathsf{pre}}$ and $\rightharpoonup_{\mathsf{post}}$, produce the methods for the pre- and post-condition hierarchy checkers. The judgments are given in figures 4.6, 4.7, 4.8, and 4.9.

As the **[defn$^c$]** rule and the **[defn$^i$]** rule show, each definition in the original program generates a definition and two additional classes. The first definition corresponds to the original definition, with the contracts removed and, in the case of classes, wrapper methods inserted. These wrapper methods check for pre-condition and post-condition violations, and invoke the hierarchy checkers. The elaborator inserts wrapper methods based on the types that instances of the class might have.

Consider the *Console* class of chapter 2. The elaboration adds two wrapper methods for *getMaxSize*, because instances of *Console* can have two types: *IConsole* and *Console*. The elaborator also adds two wrapper methods for *display*:

```
class Console implements IConsole {
  int getMaxSize() { ··· }
  int getMaxSize_IConsole ···
  int getMaxSize_Console ···

  void display(String s) { ··· }
  void display_IConsole ···
  void display_Console ···
}
```

Similarly, for *RunningConsole* and *PrefixedConsole*, *Elab* adds three methods, since instances of each of those classes may take on three types. Here is *RunningConsole*:

```
class RunningConsole extends Console {
  int getMaxSize() { ··· }
  int getMaxSize_IConsole ···
  int getMaxSize_Console ···
  int getMaxSize_RunningConsole ···

  void display(String s) { ··· }
  void display_IConsole ···
  void display_Console ···
  void display_RunningConsole ···
}
```

The **[wrap]** rule specifies the shape of the wrapper methods. It uses the program *P*, the class *c* where the wrapper method appears, the type $t$ at which the method is being called, and the method header $t'$ *md* $(t_1\ x_1, \cdots, t_j\ x_j)$. The wrapper method accepts the same arguments that the original method did, plus one extra argument naming the class whose program text contains the method call. The wrapper method first checks the pre-condition $e'_b$. If the check fails, it blames the calling context for not establishing the required pre-condition. If the pre-condition succeeds, the wrapper calls the pre-condition hierarchy checker for *c*. The pre-condition hierarchy checker traverses the class and interface hierarchy, making sure that each subtype is a behavioral subtype, for the pre-conditions. If the hierarchy checking succeeds, the wrapper method calls the original method. After the method returns, it saves the result in the variable *md*, checks the post-condition, $e'_a$ and calls the post-condition hierarchy checker. Like the pre-condition hierarchy checkers, the post-condition hierarchy checker ensures that each subtype is a behavioral subtype, for the post-conditions. Finally, if the post-condition checking succeeds, the wrapper method delivers the result of the wrapped method.

Additionally, the **[wrap]** rule rewrites the contract expressions themselves so that pre- and post-condition of methods invoked by the contracts are also checked.

Here is *Console*'s *display_Console* wrapper method:[§]

```
void display_Console(String s, string cname) {
  if ( s.length() < this.getMaxSize() ) {
    (new check_Console_pre()).display(this, s);
    let { display = this.display(s) }
    in {
      (new check_Console_post())
        .display("dummy", true, this, display, s);
    }
  } else {
    preErr(cname);
  }
}
```

[§]We omit the annotations inserted by the type-checker to clarify the presentation.

The original console class's *display* method has no post-condition, so the **if**-expression from the **[wrap]** rule is eliminated. The variable *display* is bound to the result of the method, for the post-condition. The first two arguments to *check_Console_post* are initial values for accumulators and are explained below.

The second and third classes definitions introduced by the **[defn$^{\mathrm{i}}$]** and **[defn$^{\mathrm{c}}$]** rules are the hierarchy checkers. Each hierarchy checker is responsible for checking a portion of the hierarchy and combining its result with the rest of the hierarchy checkers. Unlike pre- and post-condition checking, hierarchy checking begins at the class tag for the object; it is not based on the static type of the object. As an example, consider the hierarchy diagram in figure 4.10 and this code fragment:

$$
\begin{array}{l}
I\ o\ \texttt{=}\ \mathbf{new}\ C();\\
o.m();
\end{array}
$$

When *m* is invoked, the hierarchy checkers must ensure that the hierarchy is well-formed. Since instances of *C* can never be cast to *D* or *K*, only the boxed portion of the hierarchy in figure 4.10 is checked. Thus, when *o*'s *m* is invoked, the hierarchy checking classes ensure that *I*'s pre-condition implies *C*'s pre-condition and that *J*'s pre-condition also implies *C*'s pre-condition. Similarly, when *m* returns, only *I*, *J*, and *C*'s post-conditions are checked to ensure the post-condition hierarchy is well-formed, too.

In our running console example, the following classes are generated:

- *check_IConsole_pre*,
- *check_Console_pre*,
- *check_RunningConsole_pre*,
- *check_PrefixedConsole_pre*,
- *check_IConsole_post*,
- *check_Console_post*,

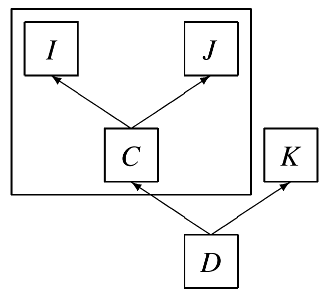



Figure 4.10: Example Hierarchy Diagram

- *check_RunningConsole_post*, and
- *check_PrefixedConsole_post*.

Each of the hierarchy-checking classes has a method for each method in the original class. The methods in the hierarchy-checking classes have the same names as the methods in the original class, although their purpose is different. The hierarchy checking methods check the pre- or post-condition for that method. Then they combine that result with the results of the rest of the hierarchy checking to determine if there are any hierarchy violations. In our example, each hierarchy checking class contains a *getMaxSize* method and a *display* method.

The **[pre$^{\mathrm{i}}$]** rule produces the pre-condition hierarchy checker for the *md* method of the interface *i*. The resulting method accepts the same arguments that *md* accepts, plus a binding for *this*. The *this* argument is passed along so the contract checking code can test the state of the object. The hierarchy checking method returns the result of the pre-condition for *md*. First, it recursively calls the hierarchy checkers for each of the immediate super-interfaces of *i* and combines their results in a disjunction. Second, it evaluates the pre-condition for *md*. Finally, the checker ensures that the hierarchy is well-formed by checking that the pre-conditions for the super-methods imply the current pre-condition. If the implication holds, the checker returns *res*, the value of this pre-condition. If the implication does not hold, the hierarchy checker method signals a hierarchy error and blames *i*, the extending interface. The rule for classes is analogous.

The pre-condition checkers for *RunningConsole* and *Console display* methods are:

```
class check_RunningConsole_pre extends Object {
  boolean display (RunningConsole this, String s) {
     let { next = (new check_Console_pre()).display(this, s)
           res = true }
     in if (!next || res)            // next ⇒ res
           res
         else
           hierErr("RunningConsole")
  }
}
```

and

```
class check_Console_pre extends Object {
  boolean display (Console this, String s) {
     let { next = (new check_IConsole_pre()).display(this, s)
           res = s.length() < this.getMaxSize() }
     in if (!next || res)            // next ⇒ res
           res
         else
           hierErr("Console")
  }
}
```

The **[post$^{i}$]** rule specifies the post-condition hierarchy checking method. The post-condition hierarchy checker is similar to the pre-condition checker. Rather than returning the truth value of each condition, however, the post-condition checker accumulates the results of the conditions in the *last* argument. Using an accumulator in this fashion means the post-condition checker uses the same recursive traversal of the type hierarchy as the pre-condition checker, but checks the implications in the reverse direction. The *tbb* argument is also an accumulator. It represents the subclass to be blamed if the implication does not hold. As mentioned above, the initial values for the accumulators *tbb* and *last* are "dummy" and **false**, respectively. Since the post-condition checker for a particular class actually blames a subclass for a hierarchy violation, the first post-condition checker never assigns blame. The initial **false** passed via *last* guarantees that no blame is assigned in the first checker and that "dummy" is ignored. Additionally, the highest class or interface in the hierarchy can never be blamed, since it cannot possibly violate the hierarchy.

Here is the code for the post-condition hierarchy checker for *getMaxSize* in both *RunningConsole* and *Console*:¶

```
class check_RunningConsole_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      RunningConsole this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)          // last ⇒ res
         (new check_Console_post())
           .getMaxSize("RunningConsole", res, this, getMaxSize)
       else
         hierErr(tbb)
  }
}
```

and

```
class check_Console_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      Console this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)          // last ⇒ res
         (new check_IConsole_post())
           .getMaxSize("Console", res, this, getMaxSize)
       else
         hierErr(tbb)
  }
}
```

Finally, the **[call$^i$]** rule shows how the elaboration re-writes method calls. Each method call becomes a call to a wrapper method, based on the type of the object whose method is invoked. For example, the code fragment:

¶In chapter 3, the *getMaxSize* methods would have been inserted directly into the *RunningConsole* and *Console* classes directly.

$$IConsole\ o = Factory.newConsole(\cdots);$$
$$o.display(\text{“It's crunch time”});$$

is rewritten to this:

$$IConsole\ o = Factory.newConsole(\cdots);$$
$$o.display\_IConsole(\text{“It's crunch time”});$$

Figure 4.11 gathers the code fragments of our running example. The left column contains the proper interfaces and classes, enriched with wrapper methods. The right column contains the hierarchy checking classes, plus the translation of the method call.

## 4.5 Evaluation

The operational semantics for Contract Java is defined as a contextual rewriting system on pairs of expressions and stores [16, 52]. Each evaluation rule has this shape:

$$P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e, \mathcal{S} \rangle \quad [\textbf{reduction rule name}]$$

A store ($\mathcal{S}$) is a mapping from variables to class-tagged field records. A field record ($\mathcal{F}$) is a mapping from field names to values. We consider configurations of expressions and stores equivalent up to $\alpha$-renaming; the variables in the store bind the free variables in the expression. Each *e* is an expression and $P$ is a program, as defined in figure 4.1.

The complete evaluation rules are in Figure 4.12. For example, the **call** rule models a method call by replacing the call expression with the body of the invoked method and syntactically replacing the formal parameters with the actual parameters. The dynamic aspect of method calls is implemented by selecting the method based on the run-time type of the object (in the store). In contrast, the *super* reduction performs **super** method selection using the class annotation that is statically determined by the type-checker.

The most noteworthy rules are **call** and **super**. Both reduce to **return** expressions. The **return** expressions are markers that signal where post-condition contract violations might occur. They are inserted by method call and super call reductions for the statement of the contract soundness theorem.

```
interface IConsole {
  int getMaxSize();
  void display(String s);
}

class Console implements IConsole {
  int getMaxSize() { ··· }
  int getMaxSize_IConsole ···
  int getMaxSize_Console ···

  void display(String s) { ··· }
  void display_IConsole ···
  void display_Console(String s, string cname) {
    if ( s.length() < this.getMaxSize() ) {
      (new check_Console_pre()).display(this, s);
      let { display = this.display(s) }
      in {
        (new check_Console_post()).display
          ("dummy", true, this, display, s);
      }
    } else {
      preErr(cname);
    }
  }
}

class RunningConsole extends Console {
  int getMaxSize_IConsole ···
  int getMaxSize_Console ···
  int getMaxSize_RunningConsole ···
  void display(String s) {
    ··· super.display
        (String.substring
          (s, ···, ··· + getMaxSize())) ···
  }
  void display_IConsole ···
  void display_Console ···
  void display_RunningConsole ···
}

class PrefixedConsole extends Console {
  int getMaxSize_IConsole ···
  int getMaxSize_Console ···
  int getMaxSize_PrefixedConsole ···
  String getPrefix() {
    return ">> ";
  }
  void display(String s) {
    super.display(this.getPrefix() + s)
  }
  void display_IConsole ···
  void display_Console ···
  void display_PrefixedConsole ···
}

class check_IConsole_pre { ··· }

class check_Console_pre extends Object {
  boolean display (Console this, String s) {
    let { next = (new check_IConsole_pre()).display(this, s)
          res = s.length() < this.getMaxSize() }
    in if (!next || res)              // next ⇒ res
         res
       else
         hierErr("Console")}}

class check_RunningConsole_pre extends Object {
  boolean display (RunningConsole this, String s) {
    let { next = (new check_Console_pre()).display(this, s)
          res = true }
    in if (!next || res)              // next ⇒ res
         res
       else
         hierErr("RunningConsole")
  }
}
class check_PrefixedConsole_pre { ··· }

class check_IConsole_post { ··· }

class check_Console_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      Console this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)              // last ⇒ res
         (new check_IConsole_post())
           .getMaxSize("Console", res, this, getMaxSize)
       else
         hierErr(tbb)
  }
}

class check_RunningConsole_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      RunningConsole this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)              // last ⇒ res
         (new check_Console_post())
           .getMaxSize("RunningConsole", res, this, getMaxSize)
       else
         hierErr(tbb)
  }
}

class check_PrefixedConsole_post { ··· }

IConsole o = ConsoleFactory(···);
o.display_IConsole("It's crunch time");
```

Figure 4.11: Elaborated Console Example

$$
\begin{array}{rcl}
e & = & \cdots \mid \mathit{object} \\
v & = & \mathit{object} \mid \textbf{null} \\
 & & \textbf{true} \mid \textbf{false}
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{E} & = & [\,] \mid \mathsf{E}\underline{\,:c\,}.\mathit{fd} \mid \mathsf{E}\underline{\,:c\,}.\mathit{fd} = e \mid v\underline{\,:c\,}.\mathit{fd} = \mathsf{E} \\
 & \mid & \mathsf{E}.\mathit{md}(e \cdots) \mid v.\mathit{md}(v \cdots \mathsf{E}\ e \cdots) \\
 & \mid & \textbf{super}\underline{\equiv v : c}\,.\mathit{md}(v \cdots \mathsf{E}\ e \cdots) \\
 & \mid & \textbf{view}\ t\ \mathsf{E} \mid \textbf{if}\ (\ \mathsf{E}\ )\ e\ \textbf{else}\ e \mid \{\ \mathsf{E}\ ;\ e\ \} \\
 & \mid & \textbf{let}\ \mathit{var} = v \cdots \mathit{var} = \mathsf{E}\ \ \mathit{var} = e \cdots \textbf{in}\ e
\end{array}
$$

$$P \vdash \langle \mathsf{E}[\mathit{object} : t.\mathit{md}(v_1, \cdots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\textbf{return} : t, c\ \{e[\mathit{object}/\mathit{this}, v_1/\mathit{var}_1, \ldots v_n/\mathit{var}_n]\}], \mathcal{S}\rangle \qquad \textbf{[call]}$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $\langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{c}_{P} c$

$$P \vdash \langle \mathsf{E}[\textbf{super}\underline{\equiv \mathit{object} : c}\,.\mathit{md}(v_1, \cdots, v_n)], \mathcal{S}\rangle \qquad \textbf{[super]}$$
$$\hookrightarrow \langle \mathsf{E}[\textbf{return} : c, c\ \{e[\mathit{object}/\mathit{this}, v_1/\mathit{var}_1, \ldots v_n/\mathit{var}_n]\}], \mathcal{S}\rangle$$
where $\langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{c}_{P} c$

$$P \vdash \langle \mathsf{E}[\textbf{return} : t, c\ \{\ v\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle \qquad \textbf{[return]}$$

---

$$P \vdash \langle \mathsf{E}[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle \qquad \textbf{[new]}$$
where $\mathit{object} \notin \mathrm{dom}(\mathcal{S})$ and $\mathcal{F} = \{c'.\mathit{fd} \mapsto \textbf{null} \mid c \leq^{c}_{P} c'$ and $\exists t$ s.t. $\langle c'.\mathit{fd}, t\rangle \in^{c}_{P} c'\}$

$$P \vdash \langle \mathsf{E}[\mathit{object}\underline{\,: c'\,}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle \qquad \textbf{[get]}$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(c'.\mathit{fd}) = v$

$$P \vdash \langle \mathsf{E}[\mathit{object}\underline{\,: c'\,}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle \qquad \textbf{[set]}$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}\rangle \qquad \textbf{[cast]}$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $c \leq_P t'$

$$P \vdash \langle \mathsf{E}[\textbf{let}\ \mathit{var}_1 = v_1 \cdots \mathit{var}_n = v_n\ \textbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v_1/\mathit{var}_1 \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle \qquad \textbf{[let]}$$

$$P \vdash \langle \mathsf{E}[\textbf{if}\ (\ \textbf{true}\ )\ e_1\ \textbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_1], \mathcal{S}\rangle \qquad \textbf{[iftrue]}$$

$$P \vdash \langle \mathsf{E}[\textbf{if}\ (\ \textbf{false}\ )\ e_1\ \textbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_2], \mathcal{S}\rangle \qquad \textbf{[iffalse]}$$

$$P \vdash \langle \mathsf{E}[\{\ v\ ;\ e\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}\rangle \qquad \textbf{[seq]}$$

$$P \vdash \langle e, \mathcal{S}'\rangle \hookrightarrow \langle e, \mathcal{S}'\rangle \qquad \textbf{[gc]}$$
where $\mathcal{S}' \subset \mathcal{S}$ and $\langle\langle e, \mathcal{S}'\rangle, \mathcal{S}'\rangle$ is closed

---

$$P \vdash \langle \mathsf{E}[\textbf{preErr}(c)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: } c \text{ violated pre-condition}, \mathcal{S}\rangle \qquad \textbf{[pre]}$$

$$P \vdash \langle \mathsf{E}[\textbf{postErr}(c)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: } c \text{ violated post-condition}, \mathcal{S}\rangle \qquad \textbf{[post]}$$

$$P \vdash \langle \mathsf{E}[\textbf{hierErr}(t)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: } t \text{ is a bad extension}, \mathcal{S}\rangle \qquad \textbf{[hier]}$$

$$P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \text{error: bad cast}, \mathcal{S}\rangle \qquad \textbf{[xcast]}$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $c \not\leq_P t'$

$$P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \textbf{null}], \mathcal{S}\rangle \hookrightarrow \langle \text{error: bad cast}, \mathcal{S}\rangle \qquad \textbf{[ncast]}$$

$$P \vdash \langle \mathsf{E}[\textbf{null}\underline{\,: c\,}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad \textbf{[nget]}$$

$$P \vdash \langle \mathsf{E}[\textbf{null}\underline{\,: c\,}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad \textbf{[nset]}$$

$$P \vdash \langle \mathsf{E}[\textbf{null}.\mathit{md}(v_1, \cdots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad \textbf{[ncall]}$$

Figure 4.12: Operational semantics for Contract Java

## 4.6 Contract Soundness

A contract monitoring tool must faithfully enforce the programmer's contracts. The contract soundness theorem guarantees this property. It relates the evaluation of the contract-elaborated program to the original program with the contracts removed. Imagine evaluation proceeding both from the program without any contracts and the contract-elaborated program:

$$
\begin{tikzcd}[column sep=small]
 & \mathit{Erase}(P) \vdash \langle \emptyset, \mathit{Erase}(e) \rangle \quad \hookrightarrow \quad \langle S_1, e_1 \rangle \quad \hookrightarrow \quad \langle S_2, e_2 \rangle \quad \cdots \\
P \arrow[ur] \arrow[dr] & \\
 & \mathit{Elab}(P) \vdash \langle \emptyset, \mathit{Elab}(e) \rangle \quad \hookrightarrow \quad \langle S'_1, e'_1 \rangle \quad \hookrightarrow \quad \langle S'_2, e'_2 \rangle \quad \cdots
\end{tikzcd}
$$

The theorem relates the top evaluation to the bottom one. Intuitively, if the program without contracts reaches a method call where the pre-condition does not hold or the hierarchy is not sound, the elaborated program must signal a corresponding pre-condition or hierarchy violation. Similarly, if the program without contracts reaches a method return where the post-condition does not hold or the hierarchy is not sound, the elaborated program must signal a corresponding post-condition or hierarchy violation.

Since contracts in our model are arbitrary Java expressions, they may have side-effects or raise errors and may thus affect the behavior of the underlying program. Considering the role of contracts as logical assertions over the state space, this is undesirable. We therefore restrict our attention to contracts that are effect-free.[‖]

DEFINITION 4.1 (EFFECT-FREE EXPRESSION). *An expression* $e$ *is* effect-free *if for any store* $S$ *such that the free variables of* $e$ *are included in dom*$(S)$*, there exists a value* $v$ *such that* $\langle e, S \rangle \hookrightarrow^* \langle v, S \rangle$.

The key to this definition is that the effect-free expressions evaluate to a value without changing the store or signalling an error. This does not mean that $e$ never allocates, however. Since garbage collection is a non-deterministic reduction step, a contract expression

[‖] In practice, there are many approaches to enforcing this restriction, each with different pros and cons.

*e* may allocate as long as the newly allocated objects are garbage when the evaluation of the contract produces a value.

In order to state the contract soundness theorem, we must give meaning to contract checking for arbitrary programs. Definition 4.2 lists the conditions that correspond to a contract violation.

DEFINITION 4.2 (CONTRACT VIOLATION).

**[pre-condition failure]**

*A program state* $\langle E[o{:}t.md(v, \cdots)], S\rangle$ *is a pre-condition contract violation if e* $\textsf{PRE}_P$ $\langle t, md\rangle$ *and* $\langle e[this/o, x/v, ...], S\rangle \hookrightarrow^* \langle \mathbf{false}, T\rangle$ *where* $\{x, ...\}$ *are the formal parameters to md as declared in* $t$ *and T is a store.*

**[post-condition failure]**

*A program state* $\langle E[\mathbf{return} : t, c\ \{v\}], S\rangle$ *is a post-condition contract violation if e* $\textsf{POST}_P$ $\langle t, md\rangle$ *and* $\langle e[this/o, \mathbf{@ret}/v], S\rangle \hookrightarrow^* \langle \mathbf{false}, T\rangle$ *for some store,* $T$.

**[pre-condition hierarchy failure]**

*A program state,* $\langle E[o{:}t.md(v, \cdots)], S\rangle$ *where* $S(o) = \langle c, \mathcal{F}\rangle$*, is a pre-condition hierarchy violation if there exist types s and s′, such that* $c \leq_P s' \prec_P s$*, and the following conditions hold:*

- *e* $\textsf{POST}_P$ $\langle s, md\rangle$,
- *e′* $\textsf{POST}_P$ $\langle s', md\rangle$,
- $\langle e[this/o, x/v, ...], S\rangle \hookrightarrow^* \langle \mathbf{false}, S\rangle$, *and*
- $\langle e'[this/o, x'/v, ...], S\rangle \hookrightarrow^* \langle \mathbf{true}, S\rangle$

*where* $\{x, ...\}$ *are the formal parameters to md as declared in* $s$*,* $\{x', ...\}$ *are the formal parameters to md as declared in* $s'$.

**[post-condition hierarchy failure]**

*A program state* $\langle E[\mathbf{return} : t, c\ \{v\}], S\rangle$ *where* $S(o) = \langle c, \mathcal{F}\rangle$ *is a pre-condition hierarchy violation if there exist types s and s′, such that* $c \leq_P s' \prec_P s$*, and these conditions all hold:*

- $e$ $\textsf{PRE}_P$ $\langle s, md \rangle$,
- $e'$ $\textsf{PRE}_P$ $\langle s', md \rangle$,
- $\langle e[\mathit{this}/o, @\mathbf{ret}/v, ...], S\rangle \hookrightarrow^* \langle \mathbf{true}, S\rangle$, *and*
- $\langle e'[\mathit{this}/o, \mathit{ret}/v, ...], S\rangle \hookrightarrow^* \langle \mathbf{false}, S\rangle$

Definition 4.3 specifies contract soundness. Intuitively, soundness guarantees that elaborated programs respect the contracts of the *original* program. More concretely, if the program that the contract elaborator produces signals an error, the evaluation of the program without contracts must reach a corresponding contract failure. If the program that the contract elaborator produces does not signal a contract error, the program without contracts must be locally contract sound at each step of its evaluation.

DEFINITION 4.3 (CONTRACT SOUNDNESS). *An elaborator Elab is* contract sound *if for any program $P$ = defn* e whose pre- and post-conditions are effect-free expressions, one of the following conditions holds:*

- $\mathit{Elab}(P) \vdash \langle \mathit{Elab}(e), \emptyset\rangle \hookrightarrow^* \langle \textsf{error: } c \textsf{ violated pre-condition}, S\rangle$
  *for some store, $S$ and class c, and*
  $\mathit{Erase}(P) \vdash \langle \mathit{Erase}(e), \emptyset\rangle \hookrightarrow^* \langle E[o.md(x, ...)], S\rangle$
  *and $\langle E[o.md(x, ...)], S\rangle$ is a pre-condition contract violation of md in c.*

- $\mathit{Elab}(P) \vdash \langle \mathit{Elab}(e), \emptyset\rangle \hookrightarrow^* \langle \textsf{error: } c \textsf{ violated post-condition}, S\rangle$
  *for some store, $S$ and class c, and*
  $\mathit{Erase}(P) \vdash \langle \mathit{Erase}(e), \emptyset\rangle \hookrightarrow^* \langle E[\mathbf{return} : t,c\ \{v\}], S\rangle$
  *and $\langle E[\mathbf{return} : t,c\ \{v\}], S\rangle$ is a post-condition violation of md in c.*

- $\mathit{Elab}(P) \vdash \langle \mathit{Elab}(e), \emptyset\rangle \hookrightarrow^* \langle \textsf{error: } t \textsf{ is a bad extension}, S\rangle$
  *for some store, $S$ and type $t$, and*
  $\mathit{Erase}(P) \vdash \langle \mathit{Erase}(e), \emptyset\rangle \hookrightarrow^* \langle e', S\rangle$
  *and $\langle e', S\rangle$ is a hierarchy violation of $t$.*

- *For each state $\langle P', S' \rangle$ such that Erase$(P) \vdash \langle$Erase$(e), \emptyset \rangle \hookrightarrow^* \langle e', S' \rangle$, $\langle e', S' \rangle$ is locally contract sound with respect to $P$ (recall that Erase$(P)$ is just $P$, but with the contract annotations removed).*

Roughly, local contract soundness for a configuration $\langle e, S \rangle$ means that in the given store $S$, the contracts about $e$ hold and that the necessary relations between contracts in $e$ hold as well. More precisely, all states that are not method calls or method returns are locally contract sound. A state that is about to evaluate a method call is locally sound if the two above conditions are true. First, the pre-condition on the method must be satisfied. Second, the pre-condition hierarchy must be behaviorally well-formed. That is, each type's pre-condition must imply each of its subtypes' pre-conditions, for the method about to be invoked. Similarly, a state that is about to perform a method return is locally sound if the post-condition on the method is satisfied and the post-condition hierarchy is behaviorally well-formed.

DEFINITION 4.4 (LOCAL CONTRACT SOUNDNESS). *A program state $\langle e, S \rangle$ is* locally contract sound *with respect to a Contract Java program $P$, if one of the following conditions holds:*

- $e = E[o.m : t\ (v_1, v_2, \cdots, v_k)]$

  **and** $S(o) = \langle c, \mathcal{F} \rangle$

  **and** *if there exists a $y$ such that $y$ $\textsf{PRE}_P$ $\langle t, m \rangle$,*
  *then $\langle y, S \rangle \hookrightarrow^* \langle \textbf{true}, U \rangle$ for some store $U$.*

  **and** *for any $s, s'$ such that $c \leq_P s \leq_P s'$,*
  *if there exists an $x$ and $x'$ such that*
  *$x$ $\textsf{PRE}_P$ $\langle s, m \rangle$, $x'$ $\textsf{PRE}_P$ $\langle s', m \rangle$,*
  *then $\langle x, S \rangle \hookrightarrow^* \langle b, T \rangle$,*
  *$\langle x', S \rangle \hookrightarrow^* \langle b', T' \rangle$, and*
  *$b' \Rightarrow b$*

- $e = E[\textbf{return} : t, c\ \{\ v\ \}]$

**and** *if there exists a $y$ such that $y$ $\textsf{POST}_P$ $\langle t, m\rangle$,*

*then $\langle y, S\rangle \hookrightarrow^* \langle \textbf{true}, U\rangle$ for some store $U$.*

**and** *for any $s, s'$ such that $c \leq_P s \leq_P s'$,*

*if there exists an $x$ and $x'$ such that*

*$x$ $\textsf{POST}_P$ $\langle s, m\rangle$, $x'$ $\textsf{POST}_P$ $\langle s', m\rangle$,*

*then $\langle x, S\rangle \hookrightarrow^* \langle b, T\rangle$,*

*$\langle x', S\rangle \hookrightarrow^* \langle b', T'\rangle$, and*

*$b \Rightarrow b'$.*

- *e is neither a method call or method return.*

THEOREM 4.1. *The elaboration Elab is contract sound.*

PROOF SKETCH. Let $P$ be a program. Assume that *Elab*$(P)$ does not signal a contract error. If *Elab*$(P)$ is to be contract sound, we must show that each reduction step of *Erase*$(P)$ is locally contract sound.

**Lemma:** *for any program, $P$, and any reduction step that Erase$(P)$ takes, Elab$(P)$ takes that same reduction (potentially with a larger context) up to the point that Elab$(P)$ raises a contract error, or until Erase$(P)$ terminates. Further, if Elab$(P)$ raises a contract error, Erase$(P)$ makes the corresponding method call or method return.* Since the elaboration does not change any expressions except method calls, *Erase*$(P)$ and *Elab*$(P)$ are synchronized, as long as there are no method calls. Let us consider the first method call. The reductions for *Erase*$(P)$ look like this:

$$\begin{array}{rcl} \textit{Erase}(P) \vdash \langle \textit{Erase}(e), \emptyset\rangle & \hookrightarrow & \cdots \\ & \hookrightarrow & \langle E[o.m : t(v_1, \cdots, v_n)]\ , S\rangle \\ & \hookrightarrow & \langle\ E[\textbf{return} : t,\, c\ b[x_1/v_1 \cdots x_n/v_n], S\rangle \end{array}$$

where *b* is the body of the method *m*.

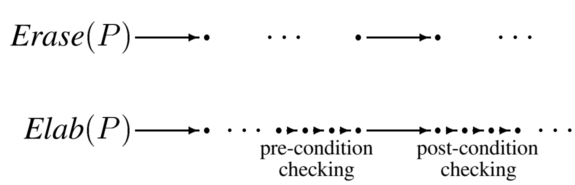
Since the ellipses do not contain any method calls, the reductions up to the first method call are identical for *Elab*$(P)$. Then, the elaborated version calls the wrapper method and the version without contracts just calls the method directly. If *Elab*$(P)$ signals a contract error or hierarchy error, the lemma holds since *Erase*$(P)$ took the same method call and the steps were synchronized. Otherwise, we know that the wrapper method does not have any effects, since their contract expressions are effect-free and *Elab*$(P)$ does not signal a pre-condition error or a hierarchy error. Thus, the reduction sequence looks like this:

$$\begin{aligned}
\mathit{Elab}(P) \vdash \langle \mathit{Elab}(e), \emptyset \rangle \;&\hookrightarrow\; \cdots \\
&\hookrightarrow\; \langle E[o.m_t(v_1, \cdots, v_n)]\ , S \rangle \\
&\hookrightarrow\; \cdots \\
&\hookrightarrow\; \langle E[F[o.m : t(v_1, \cdots, v_n)]]\ , S \rangle \\
&\hookrightarrow\; \langle\ E[F[\textbf{return} : t, c\ b[x_1/v_1 \cdots x_n/v_n]], S \rangle
\end{aligned}$$

The extra context, *F*, is the remainder of the wrapper method that checks the post-conditions and the post-condition hierarchy. By an inductive argument on the reduction sequence, we can see that, as long as *Erase*$(P)$'s reduction sequence does not contain any method returns, the lemma holds.

A similar argument applies for method returns. At the first method return, *Elab*$(P)$ will discharge the extra context it built up from the method call. This extra context corresponds to the portion of the wrapper method, after the wrapped method returns. If this code signals a contract violation, we know that *Erase*$(P)$ returns from that method. If it doesn't, the reduction sequences remain synchronized.

Pictorially, the two reduction sequences look like this:




for the first method call. The smaller arrows are the extra steps that *Elab*$(P)$ takes, before and after each method call. □

Now, using the lemma, we can prove the theorem. Assume that *Elab*$(P)$ signals a contract violation. From the lemma, we know that *Erase*$(P)$ must make the method call or method return that corresponds to the contract violation signaled from *Elab*$(P)$. From inspection of the wrapper methods, it follows that the method call or return that *Erase*$(P)$ enters is a contract violation.

Now, all that remains is to show that if *Elab*$(P)$ never signals a contract violation, *Erase*$(P)$ is locally contract sound at each step in its reduction sequence. Let $\langle e, S\rangle$ be a step in the reduction sequence starting from *Erase*$(P)$. If *e* does not decompose into some evaluation context and a method call or some evaluation context and a return instruction, it is locally hierarchy sound. Assume that it does decompose into a context and a method call. Now, we must show that the first bullet from definition 4.4 is true. Since *Elab*$(P)$ reached the same method call by the previous argument, we know that the wrapper method was invoked. From the **[wrap]** rule in figure 4.6, we can see that the pre-condition check must have succeeded. All that remains is to show this:

for any $s, s'$ such that $c \leq_P s \leq_P s'$,
if there exists an $x$ and $x'$ such that
$x \ \mathsf{PRE}_P \ \langle s, m\rangle, x' \ \mathsf{PRE}_P \ \langle s', m\rangle$,
then $\langle x, S\rangle \hookrightarrow^* \langle b, T\rangle$,
$\langle x', S\rangle \hookrightarrow^* \langle b', T'\rangle$, and
$b' \Rightarrow b$

This states that if there are two types, $s$, and $s'$ with pre-conditions $x$ and $x'$ that evaluate to $b$ and $b'$, we must have $b \Rightarrow b'$. Since the hierarchy checkers traverse the entire hierarchy checking that the pre-condition of each type implies the pre-condition of each of its subtypes, this holds. Thus, this step is locally hierarchy sound.

Similarly, if *e* decomposes into a context and a method return, *Elab*$(P)$ must also have **return**ed and the wrapper method's code must have been invoked, so this step is also locally contract sound. □

# Chapter 5

# Contracts for Higher-Order Functions

Higher-order, typed programming language implementations [2, 20, 23, 33, 51] have a static type discipline that prevents certain abuses of the language's primitive operations. For example, programs that might apply non-functions, add non-numbers, or invoke methods of non-objects are all statically rejected. Yet these languages go further. Their run-time systems dynamically prevent additional abuses of the language primitives. For example, the primitive array indexing operation aborts if it receives an out of bounds index, and the division operation aborts if it receives zero as a divisor. Together these two techniques dramatically improve the quality of software built in HOT languages.

With the advent of module languages that support type abstraction [32, 43], HOT languages empower programmers to enforce their own abstractions at the type level. These abstractions have the same expressive power that the language designer uses when specifying the language's primitives. The dynamic aspect, however, has become a second-class citizen. The programmer must manually insert dynamic checks and blame is not assigned automatically when these checks fail. Even worse, it is not always possible for the programmer to manually insert these checks because the call sites may be in unavailable modules.

This chapter empowers HOT programmers to refine the type-specifications of their abstractions with additional, dynamically enforced invariants. To that end, it presents the first assertion-based contract checker for languages with higher-order functions.

The next section discusses the challenges of contracts for higher-order functions. Section 5.2 introduces the subtleties of assigning blame for higher-order contract violations through a series of examples in Scheme [14, 25]. Section 5.3 presents $\lambda^{\mathrm{CON}}$, a typed, higher-order functional programming language with contracts. Section 5.4 specifies the meaning of $\lambda^{\text{CON}}$, and section 5.5 provides an implementation of it. Section 5.6 contains a type soundness result and proves that the implementation matches the calculus. Section 5.7 shows how to extend the calculus with function contracts whose range depends on the input to the function and section 5.8 discusses the interactions between contracts and tail recursion.

## 5.1 From First-Order Function Contracts to Higher-Order Function Contracts

In procedural languages, contracts have a simple interpretation. Consider this contract:

$f$ : **int**[$>9$] $\rightarrow$ **int**[0,99]
**val rec** $f = \lambda\ x. \cdots$

It states that the argument to $f$ must be an **int** greater than 9 and that $f$ produces an **int** between 0 and 99. To enforce this contract, a contract compiler inserts code to check that $x$ is in the proper range when $f$ is called and to check that $f$'s result is in the proper range when $f$ returns. If $x$ is not in the proper range, $f$'s caller is blamed for a contractual violation. Symmetrically, if $f$'s result is not in the proper range, the blame falls on $f$ itself. In this world, detecting contractual violations and assigning blame means merely checking appropriate predicates at well-defined points in the program's evaluation.

This straightforward mechanism for checking contracts does not generalize to languages with higher-order functions. Consider this contract:

$g$ : (**int**[$>9$] $\rightarrow$ **int**[0,99]) $\rightarrow$ **int**[0,99]
**val rec** $g = \lambda\ proc. \cdots$

The contract's domain states that $g$ accepts **int** $\rightarrow$ **int** functions and promises to apply them to **int**s larger than 9. In turn, these functions are obliged to produce **int**s between 0 and 99. The contract's range obliges $g$ to produce **int**s between 0 and 99.

Although $g$ may be given $f$, whose contract matches $g$'s domain contract, $g$ should also accept functions with stricter contracts:

$h$ : **int**[$>9$] $\rightarrow$ **int**[50,99]

**val rec** $h = \lambda\ x. \cdots$

$g(h)$,

functions without explicit contracts:

$g(\lambda\ x.\ 50)$,

functions that process external data:

*read_num* : **int**[>9] → **int**[0,99]
**val rec** *read_num* = λ *n*. ⋯ read the *n*th entry from a file ⋯

*g*(*read_num*),

and functions whose behavior depends on the context:

```
val rec dual_purpose = λ x.
  if ··· predicate on some global state ···
    then 50
    else 5000.
```

as long as the context is properly established when *g* applies its argument.

Clearly, there is no algorithm to determine whether *proc* matches its contract. Even worse, it is impossible to tell if *g* applies *proc* to **int**s greater than 9 because *g* may hand *proc* to some other function.

Additionally, higher-order functions complicate blame assignment. With first-order functions, blame assignment is directly linked to pre- and post-condition violations. A pre-condition violation is the fault of the caller and a post-condition violation is the fault of the callee. In a higher-order world, however, promises and obligations are tangled in a more complex manner, mostly due to function-valued arguments.

The key observation for higher-order contract checking is that a contract checker cannot ensure that *g*'s argument meets its contract when *g* is called. Instead, it must wait until *proc* is applied. At that point, it can ensure that *proc*'s argument is greater than 9. Similarly, when *proc* returns, it can ensure that *proc*'s result is in the range from 0 to 99. Enforcing contracts in this manner ensures that the contract violation is signalled as soon as the contract checker can prove that the contract has indeed been violated. The proof takes the form of a first-order witness to the violation. Additionally, the witness enables the contract checker to properly assign blame for the violation.

## 5.2 Example Contracts

This section contains a series of Scheme examples that explain how contracts are written and the difficulties of checking them. The first few examples illustrate the syntax and the basic principles of contract checking. Sections 5.2.2 and 5.2.3 discuss the problems of contract checking in a higher-order world. Section 5.2.4 explains why it is important for contracts to be first-class values. Section 5.2.5 demonstrates how contracts can help with callbacks, the most common use of higher-order functions in a stateful world. The sections also include examples from the DrScheme [12] code base, demonstrating that each issue is important in practice.

### 5.2.1 Contracts: A First Look

Our first example is the *sqrt* function:

```
;; sqrt : number → number
(define/contract sqrt
  ((λ (x) (≥ x 0)) ⟼ (λ (x) (≥ x 0)))
  (λ (x) ···))
```

Following the tradition of *How to Design Programs* [10], the *sqrt* function is proceeded by an ML-like [43] type specification. Like Scheme's **define**, a **define/contract** expression consists of a variable and an expression for its initial value, a function in this case. In addition, the second subexpression of **define/contract** specifies a contract for the variable.

Contract expressions are either simple predicates or function contracts. Function contracts, in turn, consist of a pair of contracts, one for the domain of the function and one for the range of the function:

$CD \longmapsto CR.$

The domain portion of *sqrt*'s contract ensures that it always receives a positive number. The range portion of the contract guarantees that the result is bigger than zero. The example illustrates that, in general, contracts check only certain aspects of a function's behavior, rather than the complete semantics of the function.

The contract position of a definition can be an arbitrary expression that evaluates to a contract. This allows us to improve the contract on *sqrt* by defining a *bigger-than-zero?* predicate and using it in the definition of *sqrt*'s contract:

```
;; bigger-than-zero? : number → boolean
(define bigger-than-zero? (λ (x) (≥ x 0)))

;; sqrt : number → number
(define/contract sqrt
  (bigger-than-zero? ⟼ bigger-than-zero?)
  (λ (x) ···))
```

The contract on *sqrt* can be strengthened by making sure that its result properly relates to its argument. The dependent function contract constructor allows the programmer to specify range contracts that depend on the values of the arguments. This constructor is similar to $\longmapsto$, except that the range position of the contract is not simply a contract. Instead, it is a function that accepts the arguments to the original function and returns a contract:

$$CD \overset{d}{\longmapsto} (\lambda\ (arg)\ CR)$$

Here is an example of a dependent contract for *sqrt*:

```
;; sqrt : number → number
(define/contract sqrt
  (bigger-than-zero? ⟼d
   (λ (x)
     (λ (res)
       (and (bigger-than-zero? x)
            (≤ (abs (− x (∗ res res))) 0.01)))))
  (λ (x) ···))
```

This contract, in addition to stating that the result of *sqrt* is positive, also ensures that the square of the result is within 0.01 of the argument.

```
(module preferences scheme/contract
  (provide add-panel open-dialog)

  ;; add-panel : (panel → panel) → void
  (define/contract add-panel
    ((any ⟼
      (λ (new-child)
        (let ([children (send (send new-child get-parent)
                              get-children)])
          (eq? (car children) new-child)))
     ⟼ any)
    (λ (make-panel)
      (set! make-panels (cons make-panel make-panels))))

  ;; make-panels : (listof (panel → panel))
  (define make-panels null)

  ;; open-dialog : → void
  (define open-dialog
    (λ ()
      (let∗ ([d (make-object dialog%)]
             [sp (make-object single-panel% d)]
             [children (map (call-make-panel sp) make-panels)])
        · · ·)))

  ;; call-make-panel : panel → (panel → panel) → panel
  (define call-make-panel
    (λ (sp)
      (λ (make-panel)
        (make-panel sp)))))
```

Figure 5.1: Contract specified with *add-panel*

### 5.2.2 Enforcement at First-Order Types

The key to checking an assertion contract on a higher-order function is to postpone the contract enforcement until the higher-order function receives a flat value as an argument or produces a flat value as a result. This section demonstrates why these delays are necessary and discusses some ramifications of delaying the contracts. Consider the following toy module:

```
(module delayed scheme/contract
  (provide save use)

  ;; saved : integer → integer
  (define saved (λ (x) 50))

  ;; save : (integer → integer) → void
  (define/contract save
    ((bigger-than-zero? → bigger-than-zero?) → any)
    (λ (f) (set! saved f)))

  ;; use : integer → integer
  (define use
    (bigger-than-zero? → bigger-than-zero?)
    (λ (n) (saved n))))
```

The module declaration* consists of a name for the module, the language that the module is written in, a **provide** declaration and a series of definitions. This module provides *save* and *use*. The variable *saved* holds a function that is supposed to map positive numbers to positive numbers. Since it is not exported from the module, it has no contract. The getter (*use*) and setter (*save*) are the two visible accessors of *saved*. The function *save* stores a new function and *use* invokes the saved function. Naturally, it is impossible for *save* to detect if the value of *saved* will always be applied to positive numbers since it cannot determine every argument to *use*. Worse, *save* cannot guarantee that each time *saved*'s value is applied that it will return a positive result. Thus, the contract checker delays the enforcement of *save*'s contract until *save*'s argument is actually applied and returns. Accordingly, violations of *save*'s contract might not be detected until *use* is called.

In general, a higher-order contract checker must be able to track contracts during evaluation from the point where the contract is established (the call site for *save*) to the discovery of the contract violation (the return site for *use*), potentially much later in the evaluation. To assign blame, the contract checker must also be able to report both where the violation was discovered and where the contract was established.

*For details of the module language used here, see the MzScheme manual [14].

```
(module preferences scheme
  (provide add-panel open-dialog)

  ;; add-panel : (panel → panel) → void
  (define add-panel
    (λ (make-panel)
      (set! make-panels (cons make-panel make-panels))))

  ;; make-panels : (listof (panel → panel))
  (define make-panels null)

  ;; open-dialog : → void
  (define open-dialog
    (λ ()
      (let∗ ([d (make-object dialog%)]
             [sp (make-object single-panel% d)]
             [children (map (call-make-panel sp) make-panels)])
        ···)))

  ;; call-make-panel : panel → (panel → panel) → panel
  (define call-make-panel
    (λ (sp)
      (λ (make-panel)
        (let ([new-child (make-panel sp)]
              [children (send (send new-child get-parent)
                              get-children)])
          (unless (eq? (car children) new-child)
            (contract-error make-panel))
          new-child)))))
```

Figure 5.2: Contract manually distributed

Our toy example is clearly contrived. The underlying phenomenon, however, is common. As a real world example, consider DrScheme's preferences panel. Plug-ins to DrScheme can add additional panels to the preferences dialog. To this end, extensions register callbacks that add new panels containing GUI controls (buttons, list-boxes, pop-up menus, etc.) to the preferences dialog.

Every GUI control needs two values: a parent for the control, and a callback that is invoked when the control is manipulated. Some GUI controls need additional control-specific values, such as a label or a list of choices. In order to add new preference panels, extensions define a function that accepts a parent panel, creates a sub-panel of the parent panel, fills the sub-panel with controls that configure the extension, and returns the sub-panel. These functions are then registered by calling *add-panel*. Each time the user chooses DrScheme's preferences menu item, DrScheme constructs the preferences dialog from the registered functions.

The contract on *add-panel* ensures that *add-panel*'s arguments are functions. In addition, it guarantees that the result of each call to its argument is the first child in its parent panel. This ensures that the ordering of the preferences dialog's children panels corresponds to the order of the calls to *make-panel*.

Figure 5.1 shows the code that implements *add-panel* and *open-dialog*, with the boxed contract attached to the definition of *add-panel*. The body of *add-panel* saves the panel making function in a list. Later, when the user opens the preferences dialog, the *make-panel* functions are called and the contracts are checked, in the context of *call-make-panel*.

In comparison, figure 5.2 contains the checking code, written as if there were no higher-order contract checking. The boxed portion of the figure, excluding the inner box, is the contract checking code. The code that enforces the contracts is co-mingled with the code that implements the preferences dialog. Co-mingling these two decreases the readability of both the contract and *call-make-panel*, since client programmers now need to determine which portion of the code is the contract checking and which portion of the code is performing the work of the function. In addition, the author of the *preferences* module must find every call-site for each higher-order function. Finding these sites in general is impossible, and in practice the call sites are often in collaborators' code, whose source might not be available.

### 5.2.3 Blame and Contravariance

Assigning blame for contractual violations in the world of first-class functions is complex. The boundaries between cooperating components are more obscure than in the world with only first-order functions. In addition to invoking a component's exported functions, one component may invoke a function passed to it from another component. Applying such first-class functions corresponds to a flow of values between components. Accordingly, the blame for a corresponding contract violation must lie with the supplier of the bad value, no matter if the bad value was passed by directly applying an exported function or by applying a first-class function.

Consider the abstract example from the introduction again, but with a little more detail. Imagine that the body of $g$ is a call to $f$ with 0:

```
;; g : (integer → integer) → integer
(define/contract g
  ((greater-than-nine? ⟼ between-zero-and-ninety-nine?)
   ⟼
   between-zero-and-ninety-nine?)
  (λ (f) (f 0)))
```

At the point when $g$ invokes $f$, the *greater-than-nine?* portion of $g$'s contract fails. This is a violation of $g$'s domain contract which, in a first-order world, is the fault of $g$'s caller. But in this case, the blame for the violation must lie with $g$ itself, since $g$ promises to apply its argument only to values greater than 9. This reversal of blame is due to the contra-variance of function application and occurs because functions are first-class values.

Imagine a variation of the above example where $g$ applies $f$ to 10 instead of 0. Further, imagine that $f$ returns $-10$. If $g$ were to return that value, it would be blamed for producing a bad result. Rather than blame $g$, however, the contract on $f$'s result is checked before $g$ returns. Accordingly, the $-10$ would trigger a contract violation error because $g$'s contract obliges its callers to supply functions that produce numbers greater than 9. In this case, $f$ would be blamed for supplying a bad value to $g$.

As with first-order function contract checking, two parties are involved for each contract: the function and its caller. Unlike first-order function contract checking, a more general rule applies for blame assignment. The function itself is responsible for the positive positions (covariant positions) of the contract and the function's caller is responsible for negative positions (contravariant positions) of the contract. This means that the contract enforcement mechanism must be able to track the negative and positive positions of a contract to determine which party is to blame.

This problem of assigning blame naturally appears in contracts from DrScheme's implementation. For example, DrScheme creates a separate thread to evaluate user's programs. Typically, extensions to DrScheme need to initialize thread-specific hidden state before the user's program is run. The accessors and mutators for this state implicitly accept the current thread as a parameter, so the code that initializes the state must run on the user's thread.[†]

To enable DrScheme's extensions to run code on the user's thread, DrScheme provides the primitive *run-on-user-thread*. It accepts a thunk, queues the thunk to be run on the user's thread and returns. It has a contract that promises that when the argument thunk is applied, the current thread is the user's thread:

```
(define/contract run-on-user-thread
  (((λ () (eq? (current-thread) user-thread)) ⟼ any)
   ⟼
   any)
  (λ (thunk)
    ···))
```

This contract is a higher-order function contract. It only has one interesting aspect: the precondition of the function passed to *run-on-user-thread*. This is a covariant position of the function contract (since the contravariant position of a contravariant position is again covariant) which, according to our rule for blame assignment, means that *run-on-user-thread* is responsible for establishing this contract. Therefore, *run-on-user-thread* contractually

[†]This state is not available to user's program because the accessors and mutators are not lexically available to the user's program.

;; *make/c* : $(\alpha\ \alpha \to$ **bool**$) \to \alpha \to \alpha \to$ **bool**
(**define** (*make/c op*) ($\lambda$ ($x$) ($\lambda$ ($y$) (*op y x*))))

;; *>=/c*, *<=/c* : **number** $\to$ **number** $\to$ **bool**
(**define** *>=/c* (*make/c* $\geq$))
(**define** *<=/c* (*make/c* $\leq$))

;; *eq/c*, *equal/c* : **any** $\to$ **any** $\to$ **bool**
(**define** *eq/c* (*make/c eq?*))
(**define** *equal/c* (*make/c equal?*))

;; *any* : **any** $\to$ **bool**
(**define** *any* ($\lambda$ ($x$) #t))

Figure 5.3: Abstraction for Predicate Contracts

promises clients of this function that the thunks they supply are applied on the user's thread. Thus, these thunks can initialize the user's thread's state.

### 5.2.4 First-class Contracts

Experience with DrScheme has shown that certain patterns of contracts recur frequently. To support abstraction over these patterns, contracts are values that can be passed to and returned from functions. For example, curried versions of comparison operators are commonly used (see figure 5.3).

More interestingly, certain patterns of higher-order function contracts are also common. As an example, it is common in DrScheme to pass mixins [13, 17] as values. In DrScheme, a mixin is a function that accepts a class and returns a class derived from its argument. Since extensions of DrScheme supply mixins to DrScheme, it is important to ensure that the result of a mixin is, in fact, derived from its input. Since this contract is so common, it is defined as a part of DrScheme's contract library:

;; *mixin-contract* : (**class** $\to$ **class**) **contract**
(**define** *mixin-contract*
  (*class?* $\overset{d}{\longmapsto}$ ($\lambda$ (*arg*) ($\lambda$ (*res*) (*subclass? res arg*)))))

```
(module preferences scheme/contract
  (provide add-panel ···)
  ;; preferences:add-panel : (panel → panel) → void
  (define/contract add-panel
    ((any ⟼d
      (λ (sp)
        (let ([pre-children (copy-spine (send sp get-children))])
          (λ (new-child)
            (let ([post-children (send sp get-children)])
              (and (= (length post-children)
                      (add1 (length pre-children)))
                   (andmap eq?
                           (cdr post-children)
                           pre-children)
                   (eq? (car post-children) new-child)))))))
     ⟼
     any)
    (λ (make-panel)
      (set! make-panels (cons make-panel make-panels))))

  ···)
```

Figure 5.4: Preferences panel contract, protecting the panel

This contract is a dependent contract. It states that the input to the function is a class and its result is a subclass of the input.

Further, it is common for the contracts on these mixins to guarantee that the base class passed to the mixin is not just any class, but a class that implements a particular interface. To support these contracts, DrScheme's contract library provides this function that constructs a contract:

```
;; mixin-contract/interface : interface → (class → class) contract
(define mixin-contract/interface
  (λ (interface)
    ((λ (x) (implements? x interface))
     ⟼d
     (λ (arg) (λ (res) (subclass? res arg))))))
```

The *mixin-contract/interface* function accepts an interface as an argument and produces a contract similar to *mixin-contract*, except that the contract guarantees that input to the function is a class that implements the given interface.

Although the mixin contract is, in principle, checkable by a type system, no such type system is currently implemented. OCaml [33] is rich enough to express mixins, but type-checking fails for any interesting use of mixins [31, 46]. This contract is an example where the expressiveness of contracts leads to an opportunity to improve existing type systems. Hopefully this example will encourage type system designers to build richer type systems that support practical mixins.

### 5.2.5 Callbacks and Stateful Contracts

Callbacks are notorious for causing problems in preserving invariants. Szyperski shows why callbacks are important and how they cause problems [50]. In short, code that invokes the callback must ensure that some state is not modified during the dynamic extent of the callback. Typically, this invariant is maintained by examining some state before the callback is invoked and comparing it to the state after the callback returns.‡

Consider this simple library for registering and invoking callbacks.

```
(module callbacks scheme/contract
  (provide register-callback invoke-callback)

  ;; register-callback : (→ void) → void
  (define/contract register-callback
    (any
     ↦d→
     (λ (arg)
       (let ([old-state ··· save the relevant state ···])
         (λ (res)
            ··· compare the new state to the old state ···))))
    (λ (c)
      (set! callback c)))
```

‡In practice lock variables are often used for this; the technique presented adapts *mutatis mutandis.* to a lock-variable based solution to the callback problem

```
;; invoke-callback : → void
(define invoke-callback
  (λ ()
    (callback)))

;; callback : → void
(define callback (λ () (void)))
```

The function *register-callback* accepts a callback function and registers it as the current callback. The *invoke-callback* function calls the callback. The contract on *register-callback* makes use of the dependent contract constructor in a new way. The contract checker applies the dependent contract to the *original* function's arguments before the function itself is applied. Therefore, the range portion of a dependent contract can determine key aspects of the state and save them in the closure of the resulting predicate. When that predicate is called with the result of the function and it can compare the current version of the state with the original version of the state, ensuring that the callback is well-behaved.

This technique is useful in the contract for DrScheme's preferences panel, whose contract we have already considered. Consider the revision of *add-panel*'s contract in figure 5.4. The revision does more than just ensure that the new child is the first child. In addition, it ensures that the original children of the preferences panel remain in the panel in the same order, thus preventing an extension from removing preference panels.

## 5.3 Contract Calculus

Figure 5.5 contains the syntax for the contract calculus. It is presented in a typed context to show how contracts refine types. Each program consists of a series of definitions, followed by a single expression. Each definition consists of a variable, a contract expression and an expression for initializing the variable. All of the variables bound by **val rec** in a single program must be distinct. Expressions ($M$) include abstractions, applications, variables, numbers and numeric primitives, lists and list primitives, **if** expressions, booleans, and predicates. The final expression forms specify contracts. The **contract**($S$) and $S \longmapsto S$ expressions construct flat and function contracts, respectively. A *flatp* expression returns core syntax

$$
\begin{aligned}
&P = D \cdots S \\
&D = \textbf{val rec}\ x : S = S \\
&M = \lambda\, x.M \mid M\ M \mid x \\
&\quad \mid n \mid M\ aop\ M \mid M\ rop\ M \\
&\quad \mid M{::}M \mid [] \mid hd(M) \mid tl(M) \mid mt(M) \\
&\quad \mid \textbf{if}\ M\ \textbf{then}\ M\ \textbf{else}\ M \mid \mathsf{true} \mid \mathsf{false} \mid str \\
&\quad \mid M \longmapsto M \mid \textbf{contract}(M) \\
&\quad \mid \mathit{flatp}(M) \mid \mathit{pred}(M) \mid \mathit{dom}(M) \mid \mathit{rng}(M) \mid \mathit{blame}(M) \\
&str = \texttt{""} \mid \texttt{"a"} \mid \texttt{"b"} \mid \cdots \mid \texttt{"aa"} \mid \texttt{"ab"} \mid \cdots \\
&rop = + \mid * \mid - \mid / \\
&aop = \geq \mid = \\
&x = \mathit{variables} \\
&n = 0 \mid 1 \mid \cdots \mid -1 \mid -2 \mid \cdots
\end{aligned}
$$

types

$$t = t \to t \mid t\ \textbf{list} \mid \textbf{int} \mid \textbf{bool} \mid \textbf{string} \mid t\ \textbf{contract}$$

Figure 5.5: $\lambda^{\textsc{Con}}$ Syntax and Types

true if its argument is a flat contract and false if its argument is a function contract. The *pred*, *dom*, and *rng* expressions select the fields of a contract.§ The *blame* primitive is used to assign blame to a definition that violates its contract. It aborts the program. This first model omits dependent contracts; we return to them later.

In this model, each definition is treated as if it were written by a different programmer. Thus, each definition is considered to be a separate entity for the purpose of assigning blame. In an implementation, this is too fine-grained. Blame should instead be assigned to a coarser construct, *e.g.*, Modula's modules, ML's structures and functors, or Java's packages. In DrScheme, we blame **module**s.

The types for $\lambda^{\textsc{Con}}$ are standard as in core ML (without polymorphism), plus types for contract expressions. The typing rules for contracts are given in figure 5.8. Contracts on flat

§Contracts are analogous to a **datatype** definition that has two variants, one for flat contracts and one for higher-order contracts.

evaluation contexts

$$
\begin{aligned}
D = {} & \textbf{val rec}\ x : V = V \cdots \\
& \quad \textbf{val rec}\ x : E = M \\
& \quad \textbf{val rec}\ x : M = M \cdots \\
& \quad M \\
& \mid \textbf{val rec}\ x : V = V \cdots \\
& \quad \textbf{val rec}\ x : V = E \\
& \quad \textbf{val rec}\ x : M = M \cdots \\
& \quad M \\
& \mid \textbf{val rec}\ x = V \cdots \\
& \quad E
\end{aligned}
$$

$$
\begin{aligned}
E = {} & E\ M \mid V\ E \\
& \mid E\ aop\ M \mid V\ aop\ E \mid E\ rop\ M \mid V\ rop\ E \\
& \mid E :: M \mid V :: E \mid hd(E) \mid tl(E) \\
& \mid \textbf{if}\ E\ \textbf{then}\ M\ \textbf{else}\ M \\
& \mid E \longmapsto M \mid V \longmapsto E \mid \textbf{contract}(E) \\
& \mid dom(E) \mid rng(E) \mid pred(E) \mid flatp(E) \mid blame(E) \\
& \mid \square
\end{aligned}
$$

values

$$
\begin{aligned}
V &= V :: V \mid \lambda\, x.\, M \mid str \mid n \mid \mathsf{true} \mid \mathsf{false} \\
V_d &= \textbf{val rec}\ x : V = V \cdots \\
& \quad V
\end{aligned}
$$

Figure 5.6: $\lambda^{\textsc{Con}}$ Evaluation Contexts and Values

values are tagged by the **contract** value constructor and must be predicates that operate on the appropriate type. Contracts for functions consist of two contracts, one for the domain and one for the range of the function. The typing rule for definitions mandates that the type of the contract corresponds to the type of definition. The rest of the typing rules are standard.

For example, consider this definition of the *sqrt* function:

$$
\begin{aligned}
& \textbf{val rec}\ sqrt : \textbf{contract}(\lambda\, x.x \geq \mathsf{0}) \longmapsto \textbf{contract}(\lambda\, x.x \geq \mathsf{0}) = \\
& \quad \lambda\, n. \cdots
\end{aligned}
$$

The body of the *sqrt* function has been elided. The contract on *sqrt* must be an $\longmapsto$ con-

$$
\begin{array}{rcl}
D[\ulcorner n_1 \urcorner \ / \ 0] & \longrightarrow & error(/) \\
D[\ulcorner n_1 \urcorner + \ulcorner n_2 \urcorner] & \longrightarrow & D[\ulcorner n_1 + n_2 \urcorner] \\
D[\ulcorner n_1 \urcorner * \ulcorner n_2 \urcorner] & \longrightarrow & D[\ulcorner n_1 * n_2 \urcorner] \\
D[\ulcorner n_1 \urcorner \ / \ \ulcorner n_2 \urcorner] & \longrightarrow & D[\ulcorner n_1 \ / \ n_2 \urcorner] \\
D[\ulcorner n_1 \urcorner - \ulcorner n_2 \urcorner] & \longrightarrow & D[\ulcorner n_1 - n_2 \urcorner] \\
D[\ulcorner n_1 \urcorner \geq \ulcorner n_2 \urcorner] & \longrightarrow & D[\mathsf{true}] \\
 & \multicolumn{2}{l}{\text{if } n_1 \geq n_2} \\
D[\ulcorner n_1 \urcorner \geq \ulcorner n_2 \urcorner] & \longrightarrow & D[\mathsf{false}] \\
 & \multicolumn{2}{l}{\text{if } n_1 < n_2} \\
D[\ulcorner n_1 \urcorner = \ulcorner n_2 \urcorner] & \longrightarrow & D[\mathsf{true}] \\
 & \multicolumn{2}{l}{\text{if } n_1 = n_2} \\
D[\ulcorner n_1 \urcorner = \ulcorner n_2 \urcorner] & \longrightarrow & D[\mathsf{false}] \\
 & \multicolumn{2}{l}{\text{if } n_1 \neq n_2} \\
D[\lambda\, x.M \ V] & \longrightarrow & D[M[x/V]] \\
D[x] & \longrightarrow & D[M] \\
 & \multicolumn{2}{l}{\text{where } D \text{ contains } (\textbf{define } x \ M)} \\
D[\textbf{if } \mathsf{true} \textbf{ then } M_1 \textbf{ else } M_2] & \longrightarrow & D[M_1] \\
D[\textbf{if } \mathsf{false} \textbf{ then } M_1 \textbf{ else } M_2] & \longrightarrow & D[M_2] \\
D[hd(V_1 :: V_2)] & \longrightarrow & D[V_1] \\
D[hd([])] & \longrightarrow & error(hd) \\
D[tl(V_1 :: V_2)] & \longrightarrow & D[V_2] \\
D[tl([])] & \longrightarrow & error(tl) \\
D[flatp(\textbf{contract}(V))] & \longrightarrow & D[\mathsf{true}] \\
D[flatp(V_1 \longmapsto V_2)] & \longrightarrow & D[\mathsf{false}] \\
D[pred(\textbf{contract}(V))] & \longrightarrow & D[V] \\
D[pred(V_1 \longmapsto V_2)] & \longrightarrow & error(pred) \\
D[dom(V_1 \longmapsto V_2)] & \longrightarrow & D[V_1] \\
D[dom(\textbf{contract}(V))] & \longrightarrow & error(dom) \\
D[rng(V_1 \longmapsto V_2)] & \longrightarrow & D[V_2] \\
D[rng(\textbf{contract}(V))] & \longrightarrow & error(rng) \\
D[blame(p)] & \longrightarrow & error(p)
\end{array}
$$

Figure 5.7: Reduction Semantics of $\lambda^{\text{CON}}$

tract because the type of *sqrt* is a function type. Further, the domain and range portions of the contract are predicates on integers because *sqrt* consumes and produces integers. More succinctly, the predicates embedded in this contract augment the type specification, indicating that the domain and range must be positive.

$$\frac{\Gamma + \{\, x_i = t_i, \cdots \,\} \vdash\ M_{1_i} : t_i\ \textbf{contract} \cdots \quad \Gamma + \{\, x_i = t_i, \cdots \,\} \vdash\ M_{2_i} : t_i \cdots \quad \Gamma + \{\, x_i = t_i, \cdots \,\} \vdash\ M : t}{\Gamma \vdash \textbf{val rec}\ x_i : M_{1_i} = M_{2_i} \cdots M : \langle\, t_i \cdots t \,\rangle}$$

$$\frac{\Gamma \vdash M : t \to \textbf{bool}}{\Gamma \vdash \textbf{contract}(M) : t\ \textbf{contract}}$$

$$\frac{\Gamma \vdash M_1 : t_1\ \textbf{contract} \qquad \Gamma \vdash M_2 : t_2\ \textbf{contract}}{\Gamma \vdash (M_1 \longmapsto M_2) : t_1 \to t_2\ \textbf{contract}} \qquad \frac{\Gamma \vdash E : t_1}{\Gamma \vdash \mathit{blame}(E) : t_2}$$

$$\frac{\Gamma \vdash M : t_1 \to t_2\ \textbf{contract}}{\Gamma \vdash \mathit{dom}(M) : t_1\ \textbf{contract}} \qquad \frac{\Gamma \vdash M : t_1 \to t_2\ \textbf{contract}}{\Gamma \vdash \mathit{rng}(M) : t_2\ \textbf{contract}}$$

$$\frac{\Gamma \vdash M : t\ \textbf{contract}}{\Gamma \vdash \mathit{pred}(M) : t \to \textbf{bool}} \qquad \frac{\Gamma \vdash M : t\ \textbf{contract}}{\Gamma \vdash \mathit{flatp}(M) : \textbf{bool}}$$

$$\frac{\Gamma + \{x{:}t\} \vdash M : s}{\Gamma \vdash \lambda\, x.\, M : t \to s} \qquad \frac{\Gamma \vdash M_1 : t \to s \qquad \Gamma \vdash M_2 : t}{\Gamma \vdash (M_1\ M_2) : s} \qquad \frac{}{\Gamma + \{x{:}t\} \vdash x : t}$$

$$\frac{}{\Gamma \vdash n : \textbf{int}} \qquad \frac{\Gamma \vdash M_1 : \textbf{int} \qquad \Gamma \vdash M_2 : \textbf{int}}{\Gamma \vdash M_1\ \mathit{aop}\ M_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash M_1 : \textbf{int} \qquad \Gamma \vdash M_2 : \textbf{int}}{\Gamma \vdash M_1\ \mathit{rop}\ M_2 : \textbf{int}}$$

$$\frac{\Gamma \vdash M_1 : t \qquad \Gamma \vdash M_2 : t\ \textbf{list}}{\Gamma \vdash M_1 :: M_2 : t\ \textbf{list}} \qquad \frac{}{\Gamma \vdash [\,] : t\ \textbf{list}} \qquad \frac{\Gamma \vdash M : t\ \textbf{list}}{\Gamma \vdash \mathit{mt}(M) : \textbf{bool}}$$

$$\frac{\Gamma \vdash M : t\ \textbf{list}}{\Gamma \vdash \mathit{hd}(M) : t} \qquad \frac{\Gamma \vdash M : t\ \textbf{list}}{\Gamma \vdash \mathit{tl}(M) : t\ \textbf{list}}$$

$$\frac{\Gamma \vdash M_1 : \textbf{bool} \qquad \Gamma \vdash M_1 : t \qquad \Gamma \vdash M_1 : t}{\Gamma \vdash \textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3 : t}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \mathit{str} : \textbf{string}}$$

Figure 5.8: $\lambda^{\text{CON}}$ Type Rules

Figures 5.6 and 5.7 define a conventional reduction semantics for the base language without contracts [11].

## 5.4 Contract Monitoring

As explained earlier, the contract monitor must have two properties. First, it must track higher-order functions to discover contract violations. Second, it must properly assign blame for contract violations. To this end, it must be able to track the covariant and contravariant portions of each contract.

To monitor contracts, we extend the calculus with a new form of expression, some new values, evaluation contexts and reductions rules. Figure 5.10 contains the new expression form, representing an obligation:

$$M^{M,x,x}$$

The first superscript is a contract expression that the base expression is obliged to meet. The last two are variables. The variables enable the contract monitoring system to assign blame for covariant and contravariant positions. The first variable names the party responsible for positive positions of the contract and the second variable names the party responsible for negative positions.

An implementation would add a fourth superscript, representing the source location where the contract is established. This superscript would be carried along during evaluation until a contract violation is discovered, at which point it would be reported as part of the error message.

Programmers do not write obligation expressions. Instead, contracts are extracted from the definitions and turned into obligations. To enforce this, the judgement $P$ **ok** holds when there are no obligation expressions in $P$.

Figure 5.9 shows how obligations are placed on each reference to a **val rec** defined variable. The first part of the obligation is the definition's contract expression. The first variable initially is the name of the referenced definition. The second variable initially is the name of the definition where the reference occurs (or *main* if the reference occurs in the last expression). The $\mathcal{I}$ function calls $\mathcal{I}_e$ for each expression in the program. The $\mathcal{I}_e$ function accepts a program that is being transformed, a variable that specifies blame for negative positions of obligations in the expression, a set of variables that might lexically shadow **val**

$\mathcal{I}$: $P \rightarrow P$
$\mathcal{I}(P = \textbf{val rec}\ x : M_1 = M_2 \cdots M_3) =$
$\quad \textbf{val rec}\ x : \mathcal{I}_e(P,x,\emptyset,M_1) = \mathcal{I}_e(P,x,\emptyset,M_2) \cdots$
$\quad \mathcal{I}_e(P,main,\emptyset,M_3)$

$\mathcal{I}_e$: $P \times x \times \{x\} \times M \rightarrow M$
$\mathcal{I}_e(P,n,\lambda\ y.\ M,s) = \lambda\ y.\ \mathcal{I}_e(n,P,M,s \cup \{y\})$
$\mathcal{I}_e(P,n,M_1(M_2),s) = \mathcal{I}_e(n,P,M_1,s)(\mathcal{I}_e(n,P,M_2,s))$
$$\mathcal{I}_e(P,n,x,s) = \begin{cases} x^{M,x,n} & \text{if } \mathcal{H}(P,\ x,\ M) \text{ and } x \notin s \\ x & \text{otherwise} \end{cases}$$
$\mathcal{I}_e(P,n,s,num) = num$
$\mathcal{I}_e(P,n,s,M_1\ aop\ M_2) = \mathcal{I}_e(P,n,s,M_1)\ aop\ \mathcal{I}_e(P,n,s,M_2)$
$\mathcal{I}_e(P,n,s,M_1\ aop\ M_2) = \mathcal{I}_e(P,n,s,M_1)\ rop\ \mathcal{I}_e(P,n,s,M_2)$
$\mathcal{I}_e(P,n,s,M_1 :: M_2) = \mathcal{I}_e(P,n,s,M_1) :: \mathcal{I}_e(P,n,s,M_2)$
$\mathcal{I}_e(P,n,s,[]) = []$
$\mathcal{I}_e(P,n,s,hd(M)) = hd(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,tl(M)) = tl(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,mt(M)) = mt(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,\textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3) =$
$\quad \textbf{if}\ (\mathcal{I}_e(P,n,s,M_1))\ \textbf{then}\ (\mathcal{I}_e(P,n,s,M_1))\ \textbf{else}\ (\mathcal{I}_e(P,n,s,M_1))$
$\mathcal{I}_e(P,n,s,\mathsf{true}) = \mathsf{true}$
$\mathcal{I}_e(P,n,s,\mathsf{false}) = \mathsf{false}$
$\mathcal{I}_e(P,n,s,str) = str$
$\mathcal{I}_e(P,n,s,M_1 \longmapsto M_2) = \mathcal{I}_e(P,n,s,M_1) \longmapsto \mathcal{I}_e(P,n,s,M_2)$
$\mathcal{I}_e(P,n,s,\textbf{contract}(M)) = \textbf{contract}(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,flatp(M)) = flatp(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,pred(M)) = pred(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,dom(M)) = dom(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,rng(M)) = rng(\mathcal{I}_e(P,n,s,M))$
$\mathcal{I}_e(P,n,s,blame(M)) = blame(\mathcal{I}_e(P,n,s,M))$

$\mathcal{H}(P,\ x,\ M_1)$ holds if **val rec** $x : M_1 = M_2$ is in $P$

Figure 5.9: Obligation Expression Insertion

**rec** defined variables and an expression to transform. It produces the transformed expression. The $\mathcal{H}$relation relates programs, variables, and expressions. A particular program, variable, and expression are related if the variable is defined by **val rec** in the program and the expression is the contract on the definition.

The introduction of obligation expressions induces the extension of the set of evaluation contexts. Figure 5.10 shows the new evaluation contexts. They specify that the value of the superscript in an obligation expression is determined before the base value. Additionally, the obligation expression induces a new type rule. The type rule guarantees that the obligation is an appropriate contract for the base expression.

Finally, we add a new class of values so that the calculus can express the delay in a higher-order contract. The new values are values labelled with function obligations (see figure 5.10). Although the grammar allows any value to be labelled with a function contract, the type soundness theorem coupled with the type rule for obligation expressions guarantees that the delayed values are always functions.

For the reductions in figure 5.7, superscripted evaluation proceeds just like the original evaluation, except that the superscript is carried from the instruction to its result. There are two additional reductions. First, when a predicate contract reaches a flat value, the predicate on that flat value is checked. If the predicate holds, the contract is discarded and evaluation continues. If the predicate fails, execution halts and the definition named by the variable in the positive position of the superscript is blamed.

The final reduction of figure 5.10 is the key to contract checking for higher-order functions. At an application of a superscripted procedure, the domain and range portion of the function position's superscript are moved to the argument expression and the entire application. Thus, the obligation to maintain the contract is distributed to the argument and the result of the application. The sense of positive and negative positions is reversed for the argument, ensuring that blame is properly assigned for contravariant portions of the contract.

For example, consider our definition of *sqrt* with a single use in the *main* expression. The reduction sequence for the application of *sqrt* is shown in figure 5.11. For brevity, references to variables defined by **val rec** are treated as values, even though they would actually reduce to the variable's current values. The first reduction is an example of how obligations are distributed on an application. The domain portion of the superscript contract is moved to the argument of the procedure and the range portion is moved to the application. The second reduction and the second to last reduction are examples of how flat contracts are checked. In this case, each predicate holds for each value. If, however, the predicate had failed in the second reduction step, *main* would be blamed, since *main* supplied the value to *sqrt*. If the predicate had failed in the second to last reduction step *sqrt* would be blamed, since *sqrt* produced the result.

For a second example, recall the toy higher-order program from the introduction:

$$
\begin{array}{l}
\textbf{val rec } gt9 = \lambda\ x.\ x \geq 9 \\
\textbf{val rec } bet0\_99 = \lambda\ x.\ \textbf{if } 99 \geq x \textbf{ then } x \geq 0 \textbf{ else } \mathsf{false} \\
\textbf{val rec } g : ((gt9 \longmapsto bet0\_99) \longmapsto bet0\_99) = \\
\quad \lambda\ f.\ f\ 0 \\
\\
g\ (\lambda\ x.\ 25)
\end{array}
$$

The definitions of *gt9* and *bet0_99* are merely helper functions for defining contracts and, as such, do not need contracts. Although our calculus does not allow such definitions, it is a simple extension to add them; the contract checker would simply ignore them.

Accordingly, the variable $g$ in the body of the *main* expression is the only reference to a definition with a contract. Thus, it is the only variable that is compiled into an obligation. The contract for the obligation is $g$'s contract. If a positive position of the contract is not met, $g$ is blamed and if a negative position of the contract is not met, *main* is blamed. Here is the reduction sequence:

$$
\begin{array}{l}
\quad g^{((gt9 \longmapsto bet0\_99) \longmapsto bet0\_99),g,main}\ (\lambda\ x.\ 25) \\
\longrightarrow (g\ (\lambda\ x.\ 25)^{(gt9 \longmapsto bet0\_99),main,g})^{bet0\_99,g,main} \\
\longrightarrow ((\lambda\ x.\ 25)^{(gt9 \longmapsto bet0\_99),main,g}\ 0)^{bet0\_99,g,main} \\
\longrightarrow (((\lambda\ x.\ 25)\ 0^{gt9,g,main})^{bet0\_99,main,g})^{bet0\_99,g,main} \\
\longrightarrow (((\lambda\ x.\ 25) \\
\qquad (\textbf{if } gt9\ 0 \textbf{ then } 0 \\
\qquad\quad \textbf{else } blame(g)))^{bet0\_99,main,g})^{bet0\_99,g,main} \\
\longrightarrow^* blame(g)
\end{array}
$$

In the first reduction step, the obligation on $g$ is distributed to $g$'s argument and to the result of the application. Additionally, the variables indicating blame are swapped in ($\lambda$ $x$. 25)'s obligation. The second step substitutes $\lambda\ x.\ 25$ in the body of $g$, resulting in an obligation expressions

$$M = \cdots \mid M^{M,x,x}$$

obligation type rule

$$\frac{\Gamma \vdash M_1 : t \qquad \Gamma \vdash M_2 : t\ \mathbf{contract}}{\Gamma \vdash {M_1}^{M_2,x,x} : t}$$

obligation evaluation contexts

$$E = \cdots \mid M^{\mathrm{E,x,x}} \mid E^{\mathrm{V,x,x}}$$

obligation values

$$V = \cdots \mid V^{V \longmapsto V,x,x}$$

obligation reductions

$$D[{V_1}^{\mathbf{contract}(V_2),p,n}] \xrightarrow{flat} D[\mathbf{if}\ V_2(V_1)\ \mathbf{then}\ V_1\ \mathbf{else}\ blame(\texttt{"p"})]$$

$$D[({V_1}^{(V_3 \longmapsto V_4),p,n}\ \ V_2)] \xrightarrow{hoc} D[(V_1\ \ {V_2}^{V_3,n,p})^{V_4,p,n}]$$

Figure 5.10: Monitoring Contracts in $\lambda^{\textsc{Con}}$

application of $\lambda\ x.\ \mathsf{25}$ to $\mathsf{0}$. The third step distributes the contract on $\lambda\ x.\ \mathsf{25}$ to $\mathsf{0}$ and to the result of the application. In addition, the variables for positive and negative blame switch positions again in $\mathsf{0}$'s contract. The fourth step reduces the flat contract on $\mathsf{0}$ to an **if** test that determines if the contract holds. The final reduction steps assign blame to *g* for supplying $\mathsf{0}$ to its argument, since it promised to supply a number greater than $\mathsf{9}$.

This example shows that higher-order functions and first-order functions are treated uniformly in our calculus. Higher-order functions merely require more distribution reductions that first-order functions. In fact, each nested arrow contract expression induces a distribution reduction during evaluation. For consistency, we focus on our *sqrt* example for the remainder of the chapter.

ORIGINAL PROGRAM

$\textbf{val rec}\ \mathit{sqrt} : \textbf{contract}(\lambda\ x.x \geq 0) \longmapsto \textbf{contract}(\lambda\ x.x \geq 0) =$
$\quad \lambda\ n. \cdots \text{body intentionally elided} \cdots$

$\mathit{sqrt}\ 4.0$

REDUCTIONS IN $\lambda^{\text{CON}}$

$$
\begin{aligned}
& {}_{\mathit{sqrt}}(\textbf{contract}(\lambda\ x.x \geq 0) \longmapsto \textbf{contract}(\lambda\ x.x \geq 0)),\mathit{main},\mathit{sqrt} \\
& 4.0 \\
\longrightarrow\ & (\mathit{sqrt}\ 4.0^{\textbf{contract}(\lambda\ x.x \geq 0),\mathit{main},\mathit{sqrt}})^{\textbf{contract}(\lambda\ x.x \geq 0),\mathit{sqrt},\mathit{main}} \\
\longrightarrow\ & (\mathit{sqrt}\ (\textbf{if}\ (\lambda\ x.x \geq 0)\ 4.0\ \textbf{then}\ 4.0\ \textbf{else}\ \mathit{blame}(\mathit{main})))^{\textbf{contract}(\lambda\ x.x \geq 0),\mathit{sqrt},\mathit{main}} \\
\longrightarrow^*\ & (\mathit{sqrt}\ 4.0)^{\textbf{contract}(\lambda\ x.x \geq 0),\mathit{sqrt},\mathit{main}} \\
\longrightarrow^*\ & 2.0^{\textbf{contract}(\lambda\ x.x \geq 0),\mathit{sqrt},\mathit{main}} \\
\longrightarrow\ & \textbf{if}\ (\lambda\ x.x \geq 0)\ 2.0\ \textbf{then}\ 2.0 \\
& \quad \textbf{else}\ \mathit{blame}(\mathit{sqrt}) \\
\longrightarrow^*\ & 2.0
\end{aligned}
$$

Figure 5.11: Reducing *sqrt* in $\lambda^{\text{CON}}$

## 5.5 Contract Implementation

To implement $\lambda^{\text{CON}}$, we must compile away obligation expressions. The key to the compilation is the wrapper function in figure 5.13. The wrapper function is defined in the calculus. It accepts a contract, a value to test, and two strings. These strings correspond to the variables in the superscripts.

Once *wrap* is defined, compiling the obligations is merely a matter of replacing an obligation expression with an application of *wrap*. The first argument is the contract of the referenced variable. The second argument is the expression under the obligation and the final two arguments are string versions of the variables in the obligation. Accordingly, we define a compiler ($\mathcal{C}$, as shown in figure 5.14) that maps from programs to programs. It replaces each obligation expression with the corresponding application of *wrap*.

ORIGINAL PROGRAM

$\textbf{val rec}\ \textit{sqrt} : \textbf{contract}(\lambda\ x.x \geq 0) \longmapsto \textbf{contract}(\lambda\ x.x \geq 0) =$
$\quad \lambda\ n. \cdots \text{body intentionally elided} \cdots$

*sqrt* 4.0

REDUCTIONS OF THE COMPILED EXPRESSION

```
    (wrap (contract(λ x.x ≥ 0) ⟼ contract(λ x.x ≥ 0))
          sqrt "sqrt")
    4.0
⟶* ((λ y. wrap (contract(λ x.x ≥ 0))
              (sqrt (wrap (contract(λ x.x ≥ 0))
                          y
                          "main" "sqrt"))
              "sqrt" "main")
     4.0)
⟶ wrap (contract(λ x.x ≥ 0))
        (sqrt (wrap (contract(λ x.x ≥ 0))
                    4.0
                    "main" "sqrt"))
        "sqrt" "main"
⟶ wrap (contract(λ x.x ≥ 0))
        (sqrt (if ((λ x.x ≥ 0) 4.0) then 4.0
               else blame("main")))
        "sqrt" "main"
⟶* wrap (contract(λ x.x ≥ 0)) (sqrt 4.0) "sqrt" "main"
⟶* wrap (contract(λ x.x ≥ 0)) 2.0 "sqrt" "main"
⟶ if (λ x.x ≥ 0) 2.0 then 2.0
    else blame("sqrt")
⟶* 2.0
```

Figure 5.12: Reducing *sqrt* with *wrap*

The function *wrap* is defined case-wise, with one case for each kind of contract. The first case handles flat contracts; it merely tests if the value matches the contract and blames the positive position if it doesn't. The second case of *wrap* deals with function contracts.

$$
\begin{array}{l}
\textit{wrap} : t\ \textbf{contract} \rightarrow t \rightarrow \textbf{string} \rightarrow \textbf{string} \rightarrow t \\
\textbf{val rec}\ \textit{wrap} = \lambda\ ct.\ \lambda\ x.\ \lambda\ p.\ \lambda\ n. \\
\quad \textbf{if}\ \textit{flatp}(ct)\ \textbf{then} \\
\quad\quad \textbf{if}\ (\textit{pred}(ct))\ x\ \textbf{then}\ x\ \textbf{else}\ \textit{error}(p) \\
\quad \textbf{else} \\
\quad\quad \textbf{let}\ d = \textit{dom}(ct) \\
\quad\quad\quad r = \textit{rng}(ct) \\
\quad\quad \textbf{in} \\
\quad\quad\quad \lambda\ y.\ \textit{wrap}\ r \\
\quad\quad\quad\quad\quad\quad (x\ (\textit{wrap}\ d\ y\ n\ p)) \\
\quad\quad\quad\quad\quad\quad p \\
\quad\quad\quad\quad\quad\quad n
\end{array}
$$

Figure 5.13: Contract Compiler Wrapping Function

The **let** expression is used as shorthand for two inlined applications of $\lambda$ expressions. The body of the **let** is a wrapper function that tests the original function's argument and its result by recursive calls to *wrap*. The first textual recursive call to *wrap* corresponds to the post-condition checking. It applies the range portion of the contract to the result of the original application. The second recursive call to *wrap* corresponds to the pre-condition checking. It applies the domain portion of the contract to the argument of the wrapper function. This call to *wrap* has the positive and negative blame positions reversed as befits the domain checking for a function.

Figure 5.12 shows how the compiled version of the *sqrt* program reduces. It begins with one call to *wrap* from the one obligation expression in the original program. The first reduction applies *wrap*. Since the contract in this case is a function contract, *wrap* takes the second case in its definition and returns a $\lambda$ expression. Next, the $\lambda$ expression is applied to 4.0. At this point, the function contract has been distributed to *sqrt*'s argument and to the result of *sqrt*'s application, just like the distribution reduction in $\lambda^{\text{CON}}$ (as shown in figure 5.11). The next reduction step is another call to *wrap*, in the argument to *sqrt*. This contract is flat, so the first case in the definition of *wrap* applies and the result is an **if** test. If that test had failed, the **else** branch would have assigned blame to *main* for supplying a

$\mathcal{C}(D \cdots S) = \mathcal{C}(D) \cdots \mathcal{C}(S)$
$\mathcal{C}(\textbf{val rec}\ x : S = S) = \textbf{val rec}\ x : \mathcal{C}(S) = \mathcal{C}(S)$
$\mathcal{C}(\lambda\ x.\ M) = \lambda\ x.\ \mathcal{C}(M)$
$\mathcal{C}({M_1}^{M_2,p,n}) = \mathit{wrapu}\ \mathcal{C}(M_2)\ \mathcal{C}(M_1)$ "p" "n"
$\mathcal{C}(M\ M) = \mathcal{C}(M)\ \mathcal{C}(M)$
$\mathcal{C}(x) = x$
$\mathcal{C}(n) = n$
$\mathcal{C}(M\ \mathit{aop}\ M) = \mathcal{C}(M)\ \mathit{aop}\ \mathcal{C}(M)$
$\mathcal{C}(M\ \mathit{rop}\ M) = \mathcal{C}(M)\ \mathit{rop}\ \mathcal{C}(M)$
$\mathcal{C}(M :: M) = \mathcal{C}(M) :: \mathcal{C}(M)$
$\mathcal{C}([]) = []$
$\mathcal{C}(\mathit{hd}(M)) = \mathit{hd}(\mathcal{C}(M))$
$\mathcal{C}(\mathit{tl}(M)) = \mathit{tl}(\mathcal{C}(M))$
$\mathcal{C}(\mathit{mt}(M)) = \mathit{mt}(\mathcal{C}(M))$
$\mathcal{C}(\textbf{if}\ M\ \textbf{then}\ M\ \textbf{else}\ M) = \textbf{if}\ \mathcal{C}(M)\ \textbf{then}\ \mathcal{C}(M)\ \textbf{else}\ \mathcal{C}(M)$
$\mathcal{C}(\mathsf{true}) = \mathsf{true}$
$\mathcal{C}(\mathsf{false}) = \mathsf{false}$
$\mathcal{C}(\mathit{str}) = \mathit{str}$
$\mathcal{C}(M \longmapsto M) = \mathcal{C}(M) \longmapsto \mathcal{C}(M)$
$\mathcal{C}(\mathit{flatp}(M)) = \mathit{flatp}(\mathcal{C}(M))$
$\mathcal{C}(\mathit{pred}(M)) = \mathit{pred}(\mathcal{C}(M))$
$\mathcal{C}(\mathit{dom}(M)) = \mathit{dom}(\mathcal{C}(M))$
$\mathcal{C}(\mathit{rng}(M)) = \mathit{rng}(\mathcal{C}(M))$
$\mathcal{C}(\mathit{blame}(M)) = \mathit{blame}(\mathcal{C}(M))$

Figure 5.14: Contract Compiler

bad value to *sqrt*. The test passes, however, and the **if** expression returns 4.0 in the next reduction step. After that, *sqrt* returns 2.0. Now we arrive at the final call to *wrap*. As before, the contract is a flat predicate, so *wrap* reduces to an **if** expression. This time, however, if the **if** test had failed *sqrt* would have been blamed for returning a bad result. In the final reduction, the **if** test succeeds and the result of the entire program is 2.0.

## 5.6 Correctness

The type soundness theorem for $\lambda^{\textsc{Con}}$ is standard [52].

$$
\begin{array}{rcll}
V_1{}^{(V_2 \longmapsto V_3),p,n} & \sim & \lambda\, x.\,(V_1\; x^{V_2,n,p})^{V_3,p,n} & \\
 & \sim & \textbf{val rec}\; x : M_1' = M_2' \cdots & \text{if } M_1 \sim M_1' \cdots,\, M_2 \sim M_2' \cdots, \\
 & & M' & \text{and } M \sim M' \\
\lambda\, x.\, M & \sim & \lambda\, x.\, M' & \text{if } M \sim M' \\
(M_1\; M_2) & \sim & (M_1'\; M_2') & \text{if } M_1 \sim M_1' \text{ and } M_2 \sim M_2' \\
n & \sim & n & \\
(M_1\; aop\; M_2) & \sim & (M_1'\; aop\; M_2') & \text{if } M_1 \sim M_1' \text{ and } M_2 \sim M_2' \\
(M_1\; rop\; M_2) & \sim & (M_1'\; rop\; M_2') & \text{if } M_1 \sim M_1' \text{ and } M_2 \sim M_2' \\
(M_1 :: M_2) & \sim & (M_1' :: M_2') & \text{if } M_1 \sim M_1' \text{ and } M_2 \sim M_2' \\
[] & \sim & [] & \\
hd(M) & \sim & hd(M') & \text{if } M \sim M' \\
tl(M) & \sim & tl(M') & \text{if } M \sim M' \\
\textbf{if}\; M_1 & \sim & \textbf{if}\; M_1' & \text{if } M_1 \sim M_1',\, M_2 \sim M_2' \text{, and } M_3 \sim M_3' \\
\textbf{then}\; M_2 & & \quad\textbf{then}\; M_2' & \\
\textbf{else}\; M_3 & & \quad\textbf{else}\; M_3' & \\
\mathsf{true} & \sim & \mathsf{true} & \\
\mathsf{false} & \sim & \mathsf{false} & \\
str & \sim & str & \\
M_1 \longmapsto M_2 & \sim & M_1' \longmapsto M_2' & \text{if } M_1 \sim M_1' \text{ and } M_2 \sim M_2' \\
dom(M) & \sim & dom(M') & \text{if } M \sim M' \\
rng(M) & \sim & rng(M') & \text{if } M \sim M' \\
pred(M) & \sim & pred(M') & \text{if } M \sim M' \\
flatp(M) & \sim & flatp(M') & \text{if } M \sim M' \\
blame(M) & \sim & blame(M') & \text{if } M \sim M' \\
error(x) & \sim & error(x) &
\end{array}
$$

Figure 5.15: Simulation between $\mathcal{E}_{fw}$ and $\mathcal{E}_{fw}$

THEOREM 5.1.(TYPE SOUNDNESS FOR $\lambda^{\text{CON}}$) *For any program, D, such that* $\emptyset \vdash D : \langle\, t \cdots \rangle$

*one of the following holds:*

- $D \longrightarrow^* V_d : \langle\, t \cdots \rangle$
- $D \longrightarrow^*$ *error(x), where x is either a* **val rec** *defined variable in D, /, hd, tl, pred dom, or rng, or*

- *for any $D_1$ such that $D \longrightarrow^* D_1$, there exists a $D_2$ such that $D_1 \longrightarrow D_2$. That is, $D$ diverges.*

PROOF Combine the preservation and progress lemmas for $\lambda^{\text{CON}}$. □

LEMMA 5.2. *(Preservation for $\lambda^{\text{CON}}$) If $\emptyset \vdash D : \langle\ t \cdots \rangle$ and $D_1 \longrightarrow D_2$ then $\emptyset \vdash D_2 : \langle\ t \cdots \rangle$.*

LEMMA 5.3. *(Progress for $\lambda^{\text{CON}}$) If $\emptyset \vdash D : \langle\ t \cdots \rangle$ then either $D = V_d$, or $D \longrightarrow D'$, for some $D'$.*

The remainder of this section formulates and proves a theorem that relates the evaluation of programs in the instrumented semantics from section 5.4 and the contract compiled programs from section 5.5.

To relate these two semantics, we introduce a new semantics and show how it relates to both semantics. The new semantics is an extension of the semantics given in figures 5.5, 5.6 and 5.7. In addition to those expressions it contains obligation expressions, evaluation contexts, and $\xrightarrow{\mathit{flat}}$ reduction from figure 5.10 (but not the new values in figure 5.10), and the $\xrightarrow{\mathit{wrap}}$ reduction:

$D[(\lambda\ x.\ M)^{(V_1 \longmapsto V_2),p,n}] \xrightarrow{\mathit{wrap}}$
$D[\lambda\ y.\ ((\lambda\ x.\ M)\ y^{V_1,n,p})^{V_2,p,n}]$

where $y$ is not free in $M$.

LEMMA 5.4. *The evaluators defined in figure 5.16 are partial functions that are only undefined when a program diverges.*

PROOF From an inspection of the evaluation contexts, we can prove that there is a unique decomposition of each program into an evaluation context and an instruction, unless it is a value. From this, it follows that the evaluators are functions. Moreover, by the type soundness theorem we know that they are only undefined for programs that diverge. □

THEOREM 5.5.(COMPILER CORRECTNESS)

$$\mathcal{E} = \mathcal{E}_{fh}$$

DEFINITION 5.1 (EVALUATORS). *Define* $\xrightarrow{fh}^*$ *to be the transitive closure of* $(\longrightarrow \cup \xrightarrow{flat} \cup \xrightarrow{hoc})$ *and define* $\xrightarrow{fw}^*$ *to be the transitive closure of* $(\longrightarrow \cup \xrightarrow{flat} \cup \xrightarrow{wrap})$. *The following functions are defined on programs P such that P* **ok**.

$$\mathcal{E}(P) \quad = \quad \begin{cases} <\!fn\!> & \text{if } \mathcal{C}(\mathcal{I}(P)) \longrightarrow^* \lambda\ x.\ M \\ V & \text{if } \mathcal{C}(\mathcal{I}(P)) \longrightarrow^* V \text{ and } V \neq \lambda\ x.\ M \\ error(x) & \text{if } \mathcal{C}(\mathcal{I}(P)) \longrightarrow^* error(x) \end{cases}$$

$$\mathcal{E}_{fh}(P) \quad = \quad \begin{cases} <\!fn\!> & \text{if } \mathcal{I}(P) \xrightarrow{fh}^* \lambda\ x.\ M \\ <\!fn\!> & \text{if } \mathcal{I}(P) \xrightarrow{fh}^* V^{V_2 \longmapsto V_3, p, n} \\ V & \text{if } \mathcal{I}(P) \xrightarrow{fh}^* V \text{ where} \\ & V \neq \lambda\ x.\ M \text{ and} \\ & V \neq V_1{}^{V_2 \longmapsto V_3, p, n} \\ error(x) & \text{if } \mathcal{I}(P) \xrightarrow{fh}^* error(x) \end{cases}$$

$$\mathcal{E}_{fw}(P) \quad = \quad \begin{cases} <\!fn\!> & \text{if } \mathcal{I}(P) \xrightarrow{fw}^* \lambda\ x.M \\ V & \text{if } \mathcal{I}(P) \xrightarrow{fw}^* V \text{ and } V \neq \lambda\ x.\ M \\ error(x) & \text{if } \mathcal{I}(P) \xrightarrow{fw}^* error(x) \end{cases}$$

Figure 5.16: Evaluators

PROOF Combine lemma 5.6 with lemma 5.7. □

LEMMA 5.6. $\mathcal{E} = \mathcal{E}_{fw}$

PROOF This proof establishes that the reduction sequences for $\mathcal{E}$ and for $\mathcal{E}_{fw}$ proceed in lockstep. First it shows that the evaluation contexts for any term and its compiled counterpart match and then it shows that each possible reduction in $\mathcal{E}$ is mirrored in $\mathcal{E}_{fw}$.

Except for obligations, the compiler does not change a program. Therefore, except for obligation expressions, a program and the compiled version of the program decompose into an instruction and a context identically. For obligation expressions, the compiler produces an application expression. From the definition of evaluation contexts for applications and for obligation expressions, we know that the obligation expressions and the compiled versions of obligation expressions also decompose in parallel. Accordingly, for the purposes of the proof we extend $\mathcal{C}$ as follows:

$\mathcal{C}(\Box) = \Box$

so we can write $\mathcal{C}(E[i]) = \mathcal{C}(E)[\mathcal{C}(i)]$.

Since the compiler does not change any expressions except obligations, we merely need to show that if an obligation expression is the instruction it reduces to the same expression that its compiled counterpart does. There are two cases. First, consider obligation expressions whose exponent is a flat contract:

$$E[V_1^{\textbf{contract}(V_2),p,n}]$$
$$\xrightarrow{fh} E[\textbf{if } V_2(V_1) \textbf{ then } V_1 \textbf{ else } \mathit{blame}(\texttt{"p"})]$$

The corresponding compiled expression reduces to the compiled version of the same **if** expression:

$$\begin{array}{l}
\mathcal{C}(E[V_1^{\textbf{contract}(V_2),p,n}]) \\
= \mathcal{C}(E)[\mathit{wrap}\ \textbf{contract}(\mathcal{C}(V_2))\ \mathcal{C}(V_1)\ \texttt{"p"}\ \texttt{"n"}] \\
\longrightarrow \mathcal{C}(E)[\textbf{if } \mathit{flatp}(\textbf{contract}(\mathcal{C}(V_2))) \textbf{ then} \\
\qquad\qquad \textbf{if } \mathit{pred}(\textbf{contract}(\mathcal{C}(V_2)))(\mathcal{C}(V_1)) \textbf{ then } \mathcal{C}(V_1) \textbf{ else } \mathit{blame}(\texttt{"p"}) \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{let } d = \mathit{dom}(\textbf{contract}(\mathcal{C}(V_1))) \\
\qquad\qquad\quad r = \mathit{rng}(\textbf{contract}(\mathcal{C}(V_1))) \\
\qquad\qquad \textbf{in} \\
\qquad\qquad\quad \lambda\ y.\ \mathit{wrap}\ r \\
\qquad\qquad\qquad\qquad (x\ (\mathit{wrap}\ d\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\qquad \texttt{"p"}\ \texttt{"n"}] \\
\longrightarrow \mathcal{C}(E)[\textbf{if } (\mathit{pred}(\textbf{contract}(\mathcal{C}(V_2))))(\mathcal{C}(V_1)) \textbf{ then } \mathcal{C}(V_1) \textbf{ else } \mathit{blame}(\texttt{"p"})] \\
\longrightarrow \mathcal{C}(E)[\textbf{if } \mathcal{C}(V_2)(\mathcal{C}(V_1)) \textbf{ then } \mathcal{C}(V_1) \textbf{ else } \mathit{blame}(\texttt{"p"})] \\
= \mathcal{C}(E[\textbf{if } V_2(V_1) \textbf{ then } V_1 \textbf{ else } \mathit{blame}(\texttt{"p"})])
\end{array}$$

Second, consider the result of reducing an obligation expressions whose exponent is a higher-order contract:

$$(\lambda\ x.\ M)^{V_1 \longmapsto V_2,p,n}$$
$$\xrightarrow{fh} \lambda\ y.\ ((\lambda\ x.\ M)\ y^{V_1,\texttt{"p"},\texttt{"n"}})^{V_2,\texttt{"n"},\texttt{"p"}}$$

Here is the reduction sequence for the compiled expression (with the short-hand for the **let** expression expanded):

$$\begin{array}{l}
\mathcal{C}(E[(\lambda\ x.\ M)^{V_2 \longmapsto V_3,p,n}]) \\
= \mathcal{C}(E)[\mathit{wrap}\ (\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))\ (\lambda\ x.\ \mathcal{C}(M))\ \texttt{"p"}\ \texttt{"n"}] \\
\longrightarrow \mathcal{C}(E)[\textbf{if } \mathit{flatp}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2)) \textbf{ then}
\end{array}$$

$$
\begin{array}{l}
\qquad\qquad\quad \textbf{if } (\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))(\lambda\ x.\ \mathcal{C}(M)) \textbf{ then } (\lambda\ x.\ \mathcal{C}(M)) \textbf{ else } \mathit{blame}(\texttt{"p"}) \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\quad (\lambda\ d. \\
\qquad\qquad\qquad (\lambda\ r. \\
\qquad\qquad\qquad\quad \lambda\ y.\ \mathit{wrap}\ r \\
\qquad\qquad\qquad\qquad\qquad\quad (x\ (\mathit{wrap}\ d\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{"p"}\ \texttt{"n"}) \\
\qquad\qquad\qquad \mathit{dom}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))) \\
\qquad\qquad\quad \mathit{rng}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))] \\
\longrightarrow \mathcal{C}(E)[(\lambda\ d. \\
\qquad\qquad (\lambda\ r. \\
\qquad\qquad\quad \lambda\ y.\ \mathit{wrap}\ r \\
\qquad\qquad\qquad\qquad\quad (x\ (\mathit{wrap}\ d\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\qquad\quad \texttt{"p"}\ \texttt{"n"}) \\
\qquad\qquad \mathit{rng}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))) \\
\qquad\quad \mathit{dom}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))] \\
\longrightarrow \mathcal{C}(E)[(\lambda\ d. \\
\qquad\qquad (\lambda\ r. \\
\qquad\qquad\quad \lambda\ y.\ \mathit{wrap}\ r \\
\qquad\qquad\qquad\qquad\quad (x\ (\mathit{wrap}\ d\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\qquad\quad \texttt{"p"}\ \texttt{"n"}) \\
\qquad\qquad \mathit{rng}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2))) \\
\qquad\quad \mathcal{C}(V_1)] \\
\longrightarrow \mathcal{C}(E)[(\lambda\ r. \\
\qquad\qquad \lambda\ y.\ \mathit{wrap}\ r \\
\qquad\qquad\qquad\qquad (x\ (\mathit{wrap}\ (\mathcal{C}(V_1)\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\qquad \texttt{"p"}\ \texttt{"n"}) \\
\qquad\quad\ \mathit{rng}(\mathcal{C}(V_1) \longmapsto \mathcal{C}(V_2)))] \\
\longrightarrow \mathcal{C}(E)[(\lambda\ r. \\
\qquad\qquad \lambda\ y.\ \mathit{wrap}\ r \\
\qquad\qquad\qquad\qquad (x\ (\mathit{wrap}\ (\mathcal{C}(V_1)\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\qquad \texttt{"p"}\ \texttt{"n"}) \\
\qquad\quad\ \mathcal{C}(V_2))] \\
\longrightarrow \mathcal{C}(E)[\lambda\ y.\ \mathit{wrap}\ (\mathcal{C}(V_2)) \\
\qquad\qquad\qquad\quad (x\ (\mathit{wrap}\ (\mathcal{C}(V_1))\ y\ \texttt{"n"}\ \texttt{"p"})) \\
\qquad\qquad\qquad\quad \texttt{"p"}\ \texttt{"n"}] \\
= \mathcal{C}(E[\lambda\ y.\ ((\lambda\ x.\ M)\ y^{V_1,\texttt{"p"},\texttt{"n"}})^{V_2,\texttt{"n"},\texttt{"p"}}])
\end{array}
$$

Since the above expression and the result of the $\xrightarrow{fh}$ reduction is the same, $\mathcal{E} = \mathcal{E}_{fw}$. $\square$

LEMMA 5.7. $\mathcal{E}_{fw} = \mathcal{E}_{fh}$

PROOF Intuitively, the difference between $\xrightarrow{fw}$ and $\xrightarrow{fh}$ is that the $\xrightarrow{hoc}$ reductions in $\xrightarrow{fh}$ are split into two steps for $\xrightarrow{fw}$, a $\xrightarrow{wrap}$ and an application, where the $\xrightarrow{wrap}$ reduction may come much earlier in the reduction sequence than the application.

This proof formalizes that intuition via the a simulation between $\mathcal{E}_{fh}$ and $\mathcal{E}_{fw}$, defined in figure 5.15. It relates $\xrightarrow{fw}$ reduced programs that have taken the first half of a $\xrightarrow{hoc}$ reduction with their $\xrightarrow{fh}$ counterparts. The first clause establishes the connection between sub-terms where the $\xrightarrow{wrap}$ reduction has occurred and their counterparts in the $\xrightarrow{fh}$ world.

In addition, we write that $D[M] \hat{\sim} D'[M']$ if both $D[M] \sim D'[M']$ and both $D[M]$ and $D'[M']$ are both valid decompositions, or are both values or errors.

The proof first establishes that all reductions steps match this diagram:

$$\begin{array}{ccc} M_1 & \xrightarrow{fh} & M_3 \\ \hat{\sim} & & \hat{\sim} \\ M'_1 & \xrightarrow{fw}{}^{*} & M'_3 \end{array}$$

First we consider the reductions in figure 5.7. Each of them preserves the simulation relation, so we know that $M_3 \sim M'_2$, where $M'_2$ is the term resulting by taking a single step in $\xrightarrow{fw}$ from $M'_1$. By lemma 5.8 we know that there exists $M'_3$ to satisfy the above diagram.

The only other reduction to consider is $M_1 \xrightarrow{hoc} M_3$. In this case, we have:

$$M_1 = D_1[(V_1^{V_2 \longmapsto V_3,p,n}\ V_4)]$$

and

$$M_3 = D_1[(V_1\ V_4^{V_2,n,p})^{V_3,p,n}]$$

By the definition of $\hat{\sim}$, $M'_1$ must either be:

$$D'_1[(V_1'^{V'_2 \longmapsto V'_3,p,n}\ V'_4)]$$

or

$$D'_1[((\lambda\ (y)\ (V'_1\ y^{V'_2,n,p})^{V'_3,p,n})\ V'_4)]$$

by the definition of $\sim$. In the first case, $M'_1$ reduces to the second expression by $\xrightarrow{wrap}$. The second expression reduces to

$$D'_1[((V_1\ V_4^{V_2,n,p})^{V_3,p,n})]$$

which simulates $M_3$. By lemma 5.8 we know that there exists an $M_3'$ to complete the diagram.

Finally, to prove the lemma, we must examine the overall reduction sequences by piecing together the above diagram. There are three situations to consider:

- The program runs forever under $\xrightarrow{fh}$. Clearly, by the piecing together the above diagram many times, the same program runs forever under $\xrightarrow{fw}$.
- The program reduces to an error under $\xrightarrow{fh}$. From the definition of the $\sim$ relation, we can see that the program must also reduce to the same error under $\xrightarrow{fw}$.
- The program reduces to a value under $\xrightarrow{fh}$. If the value is not a procedure, we know that the program must reduce to the same value under $\xrightarrow{fw}$, by the definition of $\sim$. If the value is a procedure, it might reduce to a different procedure, but the definitions of $\mathcal{E}_{fw}$ and $\mathcal{E}_{fh}$ identify any procedure values, and thus produce the same result.

□

LEMMA 5.8. *If $M_1 \sim M_2$, then there exists $M_3$ such that $M_1 \hat{\sim} M_3$ and $M_2 \xrightarrow{wrap}{}^* M_3$.*
PROOF If $M_1$ is a value, then $\sim$ and $\hat{\sim}$ are the same, so taking $M_3 = M_2$ completes the proof.

If $M_1$ is not a value then, by the progress lemma, it must decompose into an evaluation context and an instruction, $M_1 = E_1[i]$. Along the spine of $E_1$ are some number higher-order contract obligation values. We proceed by an inductive argument on the number of these expressions.

If there are zero such values, then $M_2$ must decompose into an evaluation context and an instruction, identically to $M_1$. This follows because the definition of $\sim$ says that the terms are structurally the same and the definition of evaluation contexts and values for $\xrightarrow{fw}$ and $\xrightarrow{fh}$ are the same if there are no higher-order contract obligation expressions in the spine of the term. So, we can just take $M_3 = M_2$.

If there are $n$ such values, then $M_2$ reduces via $\xrightarrow{wrap}$ replacing the outermost higher-order contract obligation with a $\lambda$ expression. This new term still simulates $M_1$ and has one dependent contract expressions

$$M = \cdots \mid M \overset{d}{\longmapsto} M$$

dependent contract evaluation contexts

$$E = \cdots \mid E \overset{d}{\longmapsto} M \mid V \overset{d}{\longmapsto} E$$

dependent contract reductions

$$D[V_3{}^{(V_1 \overset{d}{\longmapsto} V_2),p,n}\ V_4] \longrightarrow D[(V_3\ V_4{}^{V_1,n,p})^{(V_2\ V_4),p,n}]$$

Figure 5.17: Dependent Function Contracts for $\lambda^{\text{CON}}$

fewer higher-order contract value. Therefore, by the inductive hypothesis, we can conclude that there exists and $M_3$ such that $M_2 \overset{wrap}{\longrightarrow}{}^* M_3$ and $M_1 \hat{\sim} M_3$. $\square$

## 5.7 Dependent Contracts

Adding dependent contracts to the calculus is straightforward. The reduction relation for dependent function contracts naturally extends the reduction relation for normal function contracts. The reduction for distributing contracts at applications is the only difference. Instead of placing the range portion of the contract into the obligation, an application of the range portion to the function's original argument is placed in the obligation, as in figure 5.17.

The evaluation contexts given in figure 5.10 dictate that an obligation's superscript is reduced to a value before its base expression. In particular, this order of evaluation means that the application resulting from the dependent contract reduction in figure 5.17 is reduced before the base expression. Therefore, the procedure in the dependent contract can examine the state (of the machine) before the function proper is applied. This order of evaluation is critical for the callback examples from section 5.2.5.

## 5.8 Tail Recursion

Since the contract compiler described in section 5.5 checks post-conditions, it does not preserve tail recursion [7, 49] for procedures with post-conditions. Typically, determining if a procedure call is tail recursive is a simple syntactic test. In the presence of higher-order contracts, however, understanding exactly which calls are tail calls is a complex task. For example, consider this program:

$$\begin{array}{l} \textbf{val rec}\ \mathit{gt0}\ = \textbf{contract}(\lambda\ x.x \geq \mathsf{0}) \\ \textbf{val rec}\ f : (\mathit{gt0} \longmapsto \mathit{gt0}) \longmapsto \mathit{gt0} \\ \quad = \backslash g.\ g\ \mathsf{3} \end{array}$$

$$f\ (\lambda\ x.\ x{+}1)$$

The body of *f* is in tail position with respect to a conventional interpreter. Hence, a tail-call optimizing compiler should optimize the call to *g* and not allocate any additional stack space. But, due to the contract that *g*'s result must be larger than 0, the call to *g* cannot be optimized, according to our semantics of contract checking.[¶]

Even worse, since functions with contracts and functions without contracts can co-mingle during evaluation, sometimes a call to a function is a tail-call but at other times a call to the same function call is not a tail-call. Extending the above program, imagine that the argument to *f* was a locally defined recursive function. The recursive calls would be tails calls, since they would not be associated with any top-level variable, and thus no contract would be enforced.

Because contracts are most effective at module boundaries and experience has shown that module boundaries are typically not involved in tight loops, we conjecture that losing tail recursion for contract checking is not a problem in practice. In particular, adding these contracts to DrScheme has had no detectable effect on its performance. Removing the tail-call optimization entirely, however, would render DrScheme useless.

---

[¶]At a minimum, compiling it as a tail-call becomes much more difficult.

## 5.9 Conclusion

This chapter presents the first contract checker for higher-order functions. There are two key insights. The first is to delay the checks for function contracts until the function is applied or it returns. This allows the contract checker to enforce contracts for higher-order functions. The second insight is that blame for contract violations switches sense in the contra-positive positions of function contracts and how to track the positive and negative positions of function contracts during evaluation.

# Chapter 6

# Conclusions and Future Work

A serious impediment to a software component marketplace is assigning blame for run-time errors in a system composed of many components. Sound type systems mitigate this difficulty by statically rejecting certain incorrect component compositions. Static systems, however, are inherently limited to decidable approximations of a component's true interface requirements.

In contrast, behavioral contracts have no such limitation. Although they cannot express complete correctness specifications for a component, experience building DrScheme suggests that full correctness specifications are not particularly interesting. In particular, they are too complex and they do not improve the overall quality of the software much over merely specifing key behavioral properties. In fact, the most useful contracts are simple contracts, since programmers of client components must understand the contract to be able to program to it.

This work improves the state of the art of behavioral contract checking in three ways. First, it explains and fixes flaws in contract checking for object-oriented languages. Second, it extends contract checking to higher-order languages in a natural and intuitive manner. Finally, it lays the groundwork for a theory of contract checking, in the spirit of type checking.

I consider this work as the starting point for many interesting research directions. First, I believe that experience with behavioral contracts will reveal which contracts have the biggest impact on software quality. This information, in turn, will help focus type system research in the most fruitful directions.

Second, runtime software contract checking and static analyses like PLT's static debugger are synergistic. The static debugger improves the contract checking by validating some contracts statically; contract checking allows the static debugger to operate on each component of the program independently, dramatically increasing the size of programs that can be analyzed.

Third, trace-based debugging [6, 30] can also be improved with software contracts. Imagine a developer who buys components from several different companies, only to find that a particular component fails to live up to its contracts. Typically, merely reporting the problem is not enough for the company to be able to fix the defect. The component producer also needs test cases that reliably reproduce the problem. Such test cases help developers ensure that the problem really is with their components. Furthermore, the test cases also help them ensure that the defect is removed and does not re-appear in future releases.

Since it is not feasible for the component developer to buy every other component that might be useful in this context, finding the test cases is *de facto* the responsibility of the component consumer. Doing this, however, is not a simple matter. Typically, the programmer must extract a small test case from a very large program, often with no guidance. I believe that the combination of trace-based debugging and software contract specifications can automatically extract these test cases.

# Bibliography

[1] America, P. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[2] AT&T Bell Labratories. *Standard ML of New Jersey*, 1993.

[3] Bartetzko, D. Parallelität und Vererbung beim Programmieren mit Vertrag. Diplomarbeit, Universität Oldenburg, April 1999.

[4] Bartetzko, D., C. Fischer, M. Moller and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification*, 2001. Held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.

[5] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, june 1999.

[6] Choi, J. D., B. P. Miller and R. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[7] Clinger, W. D. Proper tail recursion and space efficiency. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.

[8] Detlefs, D. L., K. Rustan, M. Leino, G. Nelson and J. B. Saxe. Extended static checking. Technical Report 158, Compaq SRC Research Report, 1998.

[9] Duncan, A. and U. Hölzle. Adding contracts to Java with handshake. Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998.

[10] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[11] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. In *Theoretical Computer Science*, pages 235–271, 1992.

[12] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2002. To appear. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.

[13] Findler, R. B. and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, September 1998.

[14] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[15] Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[16] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proceedings of ACM Conference Principles of Programming Languages*, pages 171–183, Janurary 1998.

[17] Flatt, M., S. Krishnamurthi and M. Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

[18] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[19] Gomes, B., D. Stoutamire, B. Vaysman and H. Klawitter. *A Language Manual for Sather 1.1*, August 1996.

[20] Gosling, J., B. Joy and J. Guy Steele. *The Java(tm) Language Specification*. Addison-Wesley, 1996.

[21] Gosling, James. *The Emacs Screen Editor*. Unipress Software Inc., 1984.

[22] Holt, R. C. and J. R. Cordy. The Turing programming language. In *Communications of the ACM*, volume 31, pages 1310–1423, December 1988.

[23] Jones, M. P., A. Reid and The Yale Haskell Group. *The Hugs 98 User Manual*, 1999.

[24] Karaorman, M., U. Hölzle and J. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of Meta-Level Architectures and Reflection*, volume 1616 of *lncs*, July 1999.

[25] Kelsey, R., W. Clinger and J. R. (Editors). Revised$^5$ report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[26] Kiniry, J. R. and E. Cheong. JPP: A Java pre-processor. Technical Report CS-TR-98-15, Department of Computer Science, California Institute of Technology, 1998.

[27] Kizub, M. Kiev language specification. `http://www.forestro.com/kiev/`, 1998.

[28] Kölling, M. and J. Rosenberg. *Blue: Language Specification, version 0.94*, 1997.

[29] Kramer, R. iContract — the Java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, 1998.

[30] Larus, J. R. Abstract execution: A technique for efficiently tracing programs. *Software Practice and Experience*, 20(12):1241–1258, December 1990.

[31] Leroy, X. Manifest types, modules, and separate compilation. In *Proceedings of ACM Conference Principles of Programming Languages*, pages 109–122, Janurary 1994.

[32] Leroy, X. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM Conference Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

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

[34] Liskov, B. H. and J. Wing. Behavioral subtyping using invariants and constraints. Technical Report CMU CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999.

[35] Liskov, B. H. and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[36] Luckham, D. Programming with specifications. *Texts and Monographs in Computer Science*, 1990.

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

[38] Man Machine Systems. Design by contract for Java using JMSAssert. `http://www.mmsindia.com/DBCForJava.html`, 2000.

[39] McIlroy, M. D. Mass produced software components. In Naur, P. and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, 1968.

[40] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, 1988.

[41] Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.

[42] Milner, R. A theory of type polymorphism in programming. *Journal of Computer Systems Science*, 17:348–375, 1978.

[43] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[44] Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[45] Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.

[46] Rémy, D. and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of ACM Conference Principles of Programming Languages*, pages 40–53, January 1997.

[47] Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.

[48] Stallman, R. *GNU Emacs Manual*. Free Software Foundation Inc., 675 Mass. Ave., Cambridge, MA 02139, 1987.

[49] Steele, G. L. J. Debunking the “expensive procedure call” myth; or, Procedure call implementations considered harmful; or, LAMBDA: The ultimate goto. Technical Report 443, MIT Artificial Intelligence Laboratory, 1977. First appeared in the Proceedings of the ACM National Conference (Seattle, October 1977), 153–162.

[50] Szyperski, C. *Component Software*. Addison-Wesley, 1998.

[51] The GHC Team. *The Glasgow Haskell Compiler User’s Guide*, 1999.

[52] Wright, A. and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.