Contracts for Higher-Order Functions

Robert Bruce Findler[1] Matthias Felleisen
Northeastern University
College of Computer Science
Boston, Massachusetts 02115, USA

Abstract

Assertions play an important role in the construction of robust software. Their use in programming languages dates back to the 1970s. Eiffel, an object-oriented programming language, wholeheartedly adopted assertions and developed the "Design by Contract" philosophy. Indeed, the entire object-oriented community recognizes the value of assertion-based contracts on methods.

In contrast, languages with higher-order functions do not support assertion-based contracts. Because predicates on functions are, in general, undecidable, specifying such predicates appears to be meaningless. Instead, the functional languages community developed type systems that statically approximate interesting predicates.

In this paper, we show how to support higher-order function contracts in a theoretically well-founded and practically viable manner. Specifically, we introduce λ^{CON}, a typed lambda calculus with assertions for higher-order functions. The calculus models the assertion monitoring system that we employ in DrScheme. We establish basic properties of the model (type soundness, etc.) and illustrate the usefulness of contract checking with examples from DrScheme's code base.

We believe that the development of an assertion system for higher-order functions serves two purposes. On one hand, the system has strong practical potential because existing type systems simply cannot express many assertions that programmers would like to state. On the other hand, an inspection of a large base of invariants may provide inspiration for the direction of practical future type system research.

Categories & Subject Descriptors: D.3.3, D.2.1; **General Terms:** Design, Languages, Reliability; **Keywords:** Contracts, Higher-order Functions, Behavioral Specifications, Predicate Typing, Software Reliability

[1] Work partly conducted at Rice University, Houston TX. Address as of 9/2002: University of Chicago; 1100 E 58th Street; Chicago, IL 60637

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

1 Introduction

Dynamically enforced pre- and post-condition contracts have been widely used in procedural and object-oriented languages [11, 14, 17, 20, 21, 22, 25, 31]. As Rosenblum [27] has shown, for example, these contracts have great practical value in improving the robustness of systems in procedural languages. Eiffel [22] even developed an entire philosophy of system design based on contracts ("Design by Contract"). Although Java [12] does not support contracts, it is one of the most requested extensions.[1]

With one exception, higher-order languages have mostly ignored assertion-style contracts. The exception is Bigloo Scheme [28], where programmers can write down first-order, type-like constraints on procedures. These constraints are used to generate more efficient code when the compiler can prove they are correct and are turned into runtime checks when the compiler cannot prove them correct.

First-order procedural contracts have a simple interpretation. Consider this contract, written in an ML-like syntax:

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 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 merely means checking appropriate predicates at well-defined points in the program's evaluation.

This simple 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 must apply them to **int**s larger than 9. In turn, these functions must produce **int**s between 0 and 99. The contract's range obliges g to produce **int**s between 0 and 99.

[1] http://developer.java.sun.com/developer/bugParade/top25rfes.html

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

h : $\mathbf{int}[> 9] \to \mathbf{int}[50,99]$
val rec $h = \lambda\, x. \cdots$

$g(h)$,

functions without explicit contracts:

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

functions that process external data:

read_num : $\mathbf{int}[> 9] \to \mathbf{int}[0,99]$
val rec *read_num* $= \lambda\, n. \cdots$ read the nth entry from a file $\cdots$

g(*read_num*),

and functions whose behavior depends on the context:

val rec *dual_purpose* $= \lambda\, x.$
  if $\cdots$ predicate on some global state $\cdots$
    then 50
    else 5000.

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

Clearly, there is no algorithm to statically determine whether *proc* matches its contract, and it is not even possible to dynamically check the contract when g is applied. Even worse, it is not enough to monitor applications of *proc* that occur in g's body, because g may pass *proc* to another function or store it in a global variable.

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.

In this paper, we present a contract system for a higher-order world. The key observation 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 signaled as soon as the contract checker can establish that the contract has indeed been violated. The contract checker provides a first-order value as a witness to the contract violation. Additionally, the witness enables the contract checker to properly assign blame for the contract violation to the guilty party.

The next section introduces the subtleties of assigning blame for higher-order contract violations through a series of examples in Scheme [8, 16]. Section 3 presents λ^{CON}, a typed, higher-order functional programming language with contracts. Section 4 specifies the meaning of λ^{CON}, and section 5 provides an implementation of it. Section 6 contains a type soundness result and proves that the implementation in section 5 matches the calculus. Section 7 shows how to extend the calculus with function contracts whose range depends on the input to the function, and section 8 discusses the interactions between contracts and tail recursion.

2 Example Contracts

We begin our presentation with a series of Scheme examples that explain how contracts are written, why they are useful, and how to check them. The first few examples illustrate the syntax and the basic principles of contract checking. Sections 2.2 and 2.3 discuss the problems of contract checking in a higher-order world. Section 2.4 explains why it is important for contracts to be first-class values. Section 2.5 demonstrates how contracts can help with callbacks, the most common use of higher-order functions in a stateful world. To illustrate these points, each section also includes examples from the DrScheme [5] code base.

2.1 Contracts: A First Look

The first example is the *sqrt* function:

;; *sqrt* : **number** $\to$ **number**
(**define/contract** *sqrt*
  $((\lambda\,(x)\,(\geq x\ 0)) \longmapsto (\lambda\,(x)\,(\geq x\ 0)))$
  $(\lambda\,(x) \cdots))$

Following the tradition of *How to Design Programs* [3], the *sqrt* function is proceeded by an ML-like [23] type specification (in a comment). 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 subterm of **define/contract** specifies a contract for the variable.

Contracts are either simple predicates or function contracts. Function contracts, in turn, consist of a pair of contracts (each either a predicate or another function contract), 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 requires that it always receives a non-negative number. Similarly, the range portion of the contract guarantees that the result is non-negative. The example also 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 clarify 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** $\to$ **boolean**
(**define** *bigger-than-zero?* $(\lambda\,(x)\,(\geq x\ 0)))$

;; *sqrt* : **number** $\to$ **number**
(**define/contract** *sqrt*
  (*bigger-than-zero?* $\longmapsto$ *bigger-than-zero?*)
  $(\lambda\,(x) \cdots))$

The contract on *sqrt* can be strengthened by relating *sqrt*'s result to its argument. The dependent function contract constructor allows the programmer to specify range contracts that depend on the value of the function's argument. 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 argument to the original function and returns a contract:

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

```
(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 (instantiate dialog% () ···)]
             [sp (instantiate single-panel% () (parent 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 1. Contract Specified with *add-panel*

```
(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 (instantiate dialog% () ···)]
             [sp (instantiate single-panel% () (parent 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 2. Contract Manually Distributed

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? res)
            (≤ (abs (− x (∗ res res))) 0.01)))))
  (λ (x) ···))
```

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

2.2 Enforcement at First-Order Types

The key to checking higher-order assertion contracts is to postpone contract enforcement until some function receives a first-order value as an argument or produces a first-order value as a result. This section demonstrates why these delays are necessary and discusses some ramifications of delaying the contracts. Consider this 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** [8, 9] declaration consists of a name for the module, the language in which the module is written, a **provide** declaration and a series of definitions. This module provides *save* and *use*. The variable *saved* holds a function that should 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* is always 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.

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

```
;; make/c : (α α → bool) → α → α → bool
(define (make/c op) (λ (x) (λ (y) (op y x))))
;; ≥/c, ≤/c : number → number → bool
(define ≥/c (make/c ≥))
(define ≤/c (make/c ≤))
;; eq/c, equal/c : any → any → bool
(define eq/c (make/c eq?))
(define equal/c (make/c equal?))
;; any : any → bool
(define any (λ (x) #t))
```

Figure 3. Abstraction for Predicate Contracts

Every GUI control needs two values: a parent, 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 opens DrScheme's preferences dialog, DrScheme constructs the preferences dialog from the registered functions.

Figure 1 shows the definition of *add-panel* and its contract (boxed in the figure). The contract requires that *add-panel*'s arguments are functions that accept a single argument. In addition, the contract guarantees that the result of each call to *add-panel*'s argument is a panel and is the first child in its parent panel. Together, these checks ensure that the order of the panels in the preferences dialog matches the order of the calls to *add-panel*.

The body of *add-panel* saves the panel making function in a list. Later, when the user opens the preferences dialog, the *open-dialog* function is called, which calls the *make-panel* functions, and the contracts are checked. The *dialog%* and *single-panel%* classes are part of the primitive GUI library and **instantiate** creates instances of them.

In comparison, figure 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 concerns the contract checking and which performs the function's work. 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.

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.

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 rule is based on the number of times that each base contract appears to the left of an arrow in the higher-order contract. If the base contract appears an even number of times, the function itself is responsible for establishing the contract. If it appears an odd number of times, the function's caller is responsible. This even-odd rule captures which party supplies the values and corresponds to the standard notions of covariance (even positions) and contravariance (odd positions).

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. According to the even-odd rule, this must be *g*'s fault. In fact, *g* does supply the bad value, so *g* must be blamed.

Imagine a variation of the above example where *g* applies *f* to 10 instead of 0. Further, imagine that *f* returns -10. This is a violation of the result portion of *g*'s argument's contract and, following the even-odd rule, the fault lies with *g*'s caller. Accordingly, the contract enforcement mechanism must track the even and odd positions of a contract to determine the guilty party for contract violations.

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.[2]

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:

```
;; run-on-user-thread : (→ void) → void
(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 pre-condition of the function passed to *run-on-user-thread*. This is a covariant (even) position of the function contract which, according to the rule for blame assignment, means that *run-on-user-thread* is responsible for establishing this contract.

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

```
(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))))
  ···
  ;; copy-spine : (listof α) → (listof α)
  (define (copy-spine l) (map (λ (x) x) l)))
```

Figure 4. Preferences Panel Contract, Protecting the Panel

Therefore, *run-on-user-thread* contractually promises clients of this function that the thunks they supply are applied on the user's thread and that these thunks can initialize the user's thread's state.

2.4 First-class Contracts

Experience with DrScheme has shown that certain patterns of contracts recur frequently. To abstract over these patterns, contracts must be values that can be passed to and from functions. For example, curried comparision operators are common (see figure 3).

More interestingly, patterns of higher-order function contracts are also common. For example, DrScheme's code manipulates mixins [7, 10] as values. These mixins are functions that accept a class and returns a class derived from the argument. Since extensions of DrScheme supply mixins to DrScheme, it is important to verify that the mixin's result truly is derived from its input. Since this contract is so common, it is defined in DrScheme's contract library:

```
;; mixin-contract : (class → class) contract
(define mixin-contract
  (class? ⟼^d (λ (arg) (λ (res) (subclass? res arg)))))
```

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

The *mixin-contract/intf* 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 [18, 19, 26] and OML [26] are rich enough to express mixins, but type-checking fails for any interesting use of mixins [7], since the type system does not allow subsumption for imported classes. 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.

2.5 Callbacks and Stateful Contracts

Callbacks are notorious for causing problems in preserving invariants. Szyperski [32] shows why callbacks are important and how they cause problems. In short, code that invokes the callback must guarantee that certain state is not modified during the dynamic extent of the callback. Typically, this invariant is maintained by examining the state before the callback is invoked and comparing it to the state after the callback returns.[3]

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)))
  ;; 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, it can compare the current version of the state with the original version of the state, thus 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 4. The revision does more

[3] In practice, lock variables are often used for this; the technique presented here adapts to a lock-variable based solution to the callback problem.

core syntax

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

types

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

evaluation contexts

$$
\begin{array}{l}
P = \textbf{val rec}\ x : V = V \cdots \\
\qquad \textbf{val rec}\ x : E = e \\
\qquad d \cdots \\
\qquad e \\
\quad \mid \textbf{val rec}\ x : V = V \cdots \\
\qquad \textbf{val rec}\ x : V = E \\
\qquad d \cdots \\
\qquad e \\
\quad \mid \textbf{val rec}\ x : V = V \cdots \\
\qquad E \\
E = E\ e \mid V\ E \\
\quad \mid E\ aop\ e \mid V\ aop\ E \mid E\ rop\ e \mid V\ rop\ E \\
\quad \mid E :: e \mid V :: E \mid hd(E) \mid tl(E) \\
\quad \mid \textbf{if}\ E\ \textbf{then}\ e\ \textbf{else}\ e \\
\quad \mid E \longmapsto e \mid V \longmapsto E \mid \textbf{contract}(E) \\
\quad \mid dom(E) \mid rng(E) \mid pred(E) \mid flatp(E) \mid blame(E) \\
\quad \mid \square
\end{array}
$$

values

$$
\begin{array}{l}
V = V :: V \mid \lambda\, x.\ M \mid str \mid n \mid \mathsf{true} \mid \mathsf{false} \mid V \longmapsto V \mid \textbf{contract}(V) \\
V_p = \textbf{val rec}\ x : V = V \cdots \\
\qquad V
\end{array}
$$

Figure 5. λ^{CON} Syntax, Types, Evaluation Contexts, and Values

$P[\lceil n_1 \rceil\ /\ 0]$	$\longrightarrow$	$error(/)$
$\lceil n_1 \rceil + \lceil n_2 \rceil$	$\rightsquigarrow$	$\lceil n_1 + n_2 \rceil$
$\lceil n_1 \rceil * \lceil n_2 \rceil$	$\rightsquigarrow$	$\lceil n_1 * n_2 \rceil$
$\lceil n_1 \rceil\ /\ \lceil n_2 \rceil$	$\rightsquigarrow$	$\lceil n_1\ /\ n_2 \rceil$
$\lceil n_1 \rceil - \lceil n_2 \rceil$	$\rightsquigarrow$	$\lceil n_1 - n_2 \rceil$
$\lceil n_1 \rceil \geq \lceil n_2 \rceil$	$\rightsquigarrow$	true
	if $n_1 \geq n_2$	
$\lceil n_1 \rceil \geq \lceil n_2 \rceil$	$\rightsquigarrow$	false
	if $n_1 < n_2$	
$\lceil n_1 \rceil = \lceil n_2 \rceil$	$\rightsquigarrow$	true
	if $n_1 = n_2$	
$\lceil n_1 \rceil = \lceil n_2 \rceil$	$\rightsquigarrow$	false
	if $n_1 \neq n_2$	
$\lambda\, x.e\ V$	$\rightsquigarrow$	$e[x\ /\ V]$
$\textbf{fix}\ x.e$	$\rightsquigarrow$	$e[x\ /\ \textbf{fix}\ x.e]$
$P[x]$	$\longrightarrow$	$P[e_2]$
	where P contains $\textbf{val rec}\ x : e_1 = e_2$	
$\textbf{if}$ true $\textbf{then}\ e_1\ \textbf{else}\ e_2$	$\rightsquigarrow$	e_1
$\textbf{if}$ false $\textbf{then}\ e_1\ \textbf{else}\ e_2$	$\rightsquigarrow$	e_2
$hd(V_1 :: V_2)$	$\rightsquigarrow$	V_1
$P[hd([])]$	$\longrightarrow$	$error(hd)$
$tl(V_1 :: V_2)$	$\rightsquigarrow$	V_2
$P[tl([])]$	$\longrightarrow$	$error(tl)$
$mt([])$	$\rightsquigarrow$	true
$mt(V_1 :: V_2)$	$\rightsquigarrow$	false
$flatp(\textbf{contract}(V))$	$\rightsquigarrow$	true
$flatp(V_1 \longmapsto V_2)$	$\rightsquigarrow$	false
$pred(\textbf{contract}(V))$	$\rightsquigarrow$	V
$P[pred(V_1 \longmapsto V_2)]$	$\longrightarrow$	$error(pred)$
$dom(V_1 \longmapsto V_2)$	$\rightsquigarrow$	V_1
$P[dom(\textbf{contract}(V))]$	$\longrightarrow$	$error(dom)$
$rng(V_1 \longmapsto V_2)$	$\rightsquigarrow$	V_2
$P[rng(\textbf{contract}(V))]$	$\longrightarrow$	$error(rng)$
$P[blame(p)]$	$\longrightarrow$	$error(p)$

where $P[e] \longrightarrow P[e']$ if $e \rightsquigarrow e'$

Figure 6. Reduction Semantics of λ^{CON}

than just ensure that the new child is the first child. In addition, it guarantees that the original children of the preferences panel remain in the panel in the same order, thus preventing an extension from removing the other preference panels.

3 Contract Calculus

Although contracts can guarantee stronger properties than types about program execution, their guarantees hold only for particular program executions. In contrast, the type checker's weaker guarantees hold for *all* program executions. As such, contracts and types play synergistic roles in program development and maintenance so practical programming languages must support both. In that spirit, this calculus contains both types and contracts to show how they interact.

Figure 5 contains the syntax for the contract calculus. 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. All of the definitions are mutually recursive, except that the contract positions may only refer to defined variables that appear earlier in a program.

Expressions (e) include abstractions, applications, variables, fix points, numbers and numeric primitives, lists and list primitives, **if** expressions, booleans, and strings. The final expression forms specify contracts. The $\textbf{contract}(e)$ and $e \longmapsto e$ expressions construct flat and function contracts, respectively. A *flatp* expression returns 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.

The types for λ^{CON} are those of core ML (without polymorphism), plus types for contract expressions. The typing rules for contracts are given in figure 7. The first typing rule is for complete programs. A program's type is a record of types, written:

$$\langle\ t \cdots \rangle$$

where the first types are the types of the definitions and the last type is the type of the final expression.

Contracts on flat 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

$$\frac{\Gamma + \{\, x_j = t_j \mid 0 \leq j < i \,\} \vdash e_{1_i} : t_i \ \textbf{contract} \cdots \qquad \Gamma + \{\, x_i = t_i, \cdots \,\} \vdash e_{2_i} : t_i \cdots \qquad \Gamma + \{\, x_i = t_i, \cdots \,\} \vdash e : t}{\Gamma \vdash \textbf{val rec}\ x_i : e_{1_i} = e_{2_i} \cdots e : \langle\, t_i \cdots t \,\rangle}$$

$$\frac{\Gamma \vdash e : t \to \textbf{bool}}{\Gamma \vdash \textbf{contract}(e) : t\ \textbf{contract}} \qquad \frac{\Gamma \vdash e_1 : t_1\ \textbf{contract} \qquad \Gamma \vdash e_2 : t_2\ \textbf{contract}}{\Gamma \vdash (e_1 \longmapsto e_2) : t_1 \to t_2\ \textbf{contract}} \qquad \frac{\Gamma \vdash e : \textbf{string}}{\Gamma \vdash \mathit{blame}(e) : t}$$

$$\frac{\Gamma \vdash e : t_1 \to t_2\ \textbf{contract}}{\Gamma \vdash \mathit{dom}(e) : t_1\ \textbf{contract}} \qquad \frac{\Gamma \vdash e : t_1 \to t_2\ \textbf{contract}}{\Gamma \vdash \mathit{rng}(e) : t_2\ \textbf{contract}} \qquad \frac{\Gamma \vdash e : t\ \textbf{contract}}{\Gamma \vdash \mathit{pred}(e) : t \to \textbf{bool}} \qquad \frac{\Gamma \vdash e : t\ \textbf{contract}}{\Gamma \vdash \mathit{flatp}(e) : \textbf{bool}}$$

$$\frac{\Gamma + \{x : t1\} \vdash e : t_2}{\Gamma \vdash \lambda\, x.\ e : t_1 \to t_2} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2} \qquad \frac{}{\Gamma + \{x : t\} \vdash x : t} \qquad \frac{\Gamma + \{x : t\} \vdash e : t}{\Gamma \vdash \textbf{fix}\ x.\ e : t}$$

$$\frac{}{\Gamma \vdash n : \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1\ \mathit{aop}\ e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1\ \mathit{rop}\ e_2 : \textbf{int}}$$

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t\ \textbf{list}}{\Gamma \vdash e_1 :: e_2 : t\ \textbf{list}} \qquad \frac{}{\Gamma \vdash [\,] : t\ \textbf{list}} \qquad \frac{\Gamma \vdash e : t\ \textbf{list}}{\Gamma \vdash \mathit{mt}(e) : \textbf{bool}} \qquad \frac{\Gamma \vdash e : t\ \textbf{list}}{\Gamma \vdash \mathit{hd}(e) : t} \qquad \frac{\Gamma \vdash e : t\ \textbf{list}}{\Gamma \vdash \mathit{tl}(e) : t\ \textbf{list}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \mathit{str} : \textbf{string}}$$

Figure 7. λ^{CON} Type Rules

and one for the range of the function. The typing rule for definitions ensures that the type of the contract matches the type of the definition. The rest of the typing rules are standard.

Consider this definition of the *sqrt* function:

$$\textbf{val rec}\ \mathit{sqrt} : \textbf{contract}(\lambda\, x.x \geq 0) \longmapsto \textbf{contract}(\lambda\, x.x \geq 0) = \lambda\, n. \cdots$$

The body of the *sqrt* function has been elided. The contract on *sqrt* must be an $\longmapsto$ contract 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.[4] More succinctly, the predicates in this contract augment the *sqrt*'s type, indicating that the domain and range must be positive.

Figures 5 and 6 define a conventional reduction semantics for the base language without contracts [4].

4 Contract Monitoring

As explained earlier, the contract monitor must perform two tasks. 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 track higher-order functions through the program's evaluation and the covariant and contravariant portions of each contract.

To monitor contracts, we add a new form of expression, some new values, evaluation contexts and reduction rules. Figure 8 contains the new expression form, representing an *obligation*:

$$e^{e,x,x}$$

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 properly. The first variable names the party responsible for values that are produced by the expression under the superscript and the second variable names the party responsible for values that it consumes.

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.

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 [9].

Programmers do not write obligation expressions. Instead, contracts are extracted from the definitions and turned into obligations. To enforce this, we define the judgment p **ok** that holds when there are no obligation expressions in p.

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 is initially the name of the referenced definition. The second variable is initially the name of the definition where the reference occurs (or *main* if the reference occurs in the last expression). The function $\mathcal{I}$ (defined in the accompanying technical report [6]) specifies precisely how to insert the obligations expressions.

The introduction of obligation expressions induces the extension of the set of evaluation contexts, as shown in figure 8. 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.

[4] Technically, *sqrt* should consume and produce any number, but since λ^{CON} only contains integers and the precise details of *sqrt* are unimportant, we consider a restricted form of *sqrt* that operates on integers.

obligation expressions

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

obligation type rule

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t\ \textbf{contract}}{\Gamma \vdash {e_1}^{e_2,x,x} : t}$$

obligation evaluation contexts

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

obligation values

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

obligation reductions

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

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

Figure 8. Monitoring Contracts in λ^{CON}

Finally, we add the class of labeled values. The labels are function obligations (see figure 8). Although the grammar allows any value to be labeled with a function contract, the type soundness theorem coupled with the type rule for obligation expressions guarantees that the delayed values are always functions, or functions wrapped with additional obligations.

For the reductions in figure 6, 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 8 is the key to contract checking for higher-order functions (the *hoc* above the arrow stands for "higher-order contract"). 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. As the obligation moves to the argument position of the application, the value producer and the value consumer exchange roles. That is, values that are being provided to the function are being provided from the argument and vice versa. Accordingly, the last two superscripts of the obligation expression must be reversed, which ensures that blame is properly assigned, according to the even-odd rule.

For example, consider the definition of *sqrt* with a single use in the *main* expression. The reduction sequence for the application of *sqrt* is shown on the left in figure 10. 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

$\overline{wrap} : t\ \textbf{contract} \rightarrow t \rightarrow \textbf{string} \rightarrow \textbf{string} \rightarrow t$
$\overline{wrap} = \textbf{fix}\ wrap.\ \lambda\ ct.\ \lambda\ x.\ \lambda\ p.\ \lambda\ n.$
$\quad \textbf{if}\ flatp(ct)\ \textbf{then}$
$\quad\quad \textbf{if}\ (pred(ct))\ x\ \textbf{then}\ x\ \textbf{else}\ error(p)$
$\quad \textbf{else}$
$\quad\quad \textbf{let}\ d = dom(ct)$
$\quad\quad\quad r = rng(ct)$
$\quad\quad \textbf{in}$
$\quad\quad\quad \lambda\ y.\ wrap\ r$
$\quad\quad\quad\quad\quad (x\ (wrap\ d\ y\ n\ p))$
$\quad\quad\quad\quad\quad p$
$\quad\quad\quad\quad\quad n$

Figure 9. Contract Compiler Wrapping Function

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 higher-order program from the introduction (translated to the calculus):

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

The definitions of *gt9* and *bet0_99* are merely helper functions for defining contracts and, as such, do not need contracts. Although the 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 an even position of the contract is not met, g is blamed and if an odd position of the contract is not met, *main* is blamed. Here is the reduction sequence:

$$\begin{array}{l}
\quad g^{((gt9 \longmapsto bet0_99) \longmapsto bet0_99),g,main}\ (\lambda\ x.\ 25) \\
\longrightarrow (g\ (\lambda\ x.\ 25)^{(gt9 \longmapsto bet0_99),main,g})^{bet0_99,g,main} \\
\longrightarrow ((\lambda\ x.\ 25)^{(gt9 \longmapsto bet0_99),main,g}\ 0)^{bet0_99,g,main} \\
\longrightarrow (((\lambda\ x.\ 25)\ 0^{gt9,g,main})^{bet0_99,main,g})^{bet0_99,g,main} \\
\longrightarrow (((\lambda\ x.\ 25) \\
\quad\quad (\textbf{if}\ gt9(0)\ \textbf{then}\ 0 \\
\quad\quad\quad \textbf{else}\ blame(\texttt{"g"})))^{bet0_99,main,g})^{bet0_99,g,main} \\
\longrightarrow^{*}\ blame(\texttt{"g"})
\end{array}$$

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

ORIGINAL PROGRAM

val rec *sqrt* : **contract**($\lambda\, x.x \geq 0$) $\longmapsto$ **contract**($\lambda\, x.x \geq 0$) =
  $\lambda\, n.$ $\cdots$ body intentionally elided $\cdots$

sqrt 4

REDUCTIONS IN λ^{CON}

$$sqrt^{(\mathbf{contract}(\lambda\, x.x \geq 0) \longmapsto \mathbf{contract}(\lambda\, x.x \geq 0)),sqrt,main}\ 4$$

$$\longrightarrow (sqrt\ 4^{\mathbf{contract}(\lambda\, x.x \geq 0),main,sqrt})^{\mathbf{contract}(\lambda\, x.x \geq 0),sqrt,main}$$

$$\longrightarrow (sqrt\ (\mathbf{if}\ (\lambda\, x.x \geq 0)\ 4\ \mathbf{then}\ 4\ \mathbf{else}\ blame(main)))^{\mathbf{contract}(\lambda\, x.x \geq 0),sqrt,main}$$

$$\longrightarrow^{*} (sqrt\ 4)^{\mathbf{contract}(\lambda\, x.x \geq 0),sqrt,main}$$

$$\longrightarrow^{*} 2^{\mathbf{contract}(\lambda\, x.x \geq 0),sqrt,main}$$

$$\longrightarrow \mathbf{if}\ (\lambda\, x.x \geq 0)\ 2\ \mathbf{then}\ 2\ \mathbf{else}\ blame(sqrt)$$

$$\longrightarrow^{*} 2$$

REDUCTIONS OF THE COMPILED EXPRESSION

$$(\overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0) \longmapsto \mathbf{contract}(\lambda\, x.x \geq 0))\ sqrt\ \texttt{"sqrt"}\ \texttt{"main"})\ 4$$

$$\longrightarrow^{*} ((\lambda\, y.\ \overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ (sqrt\ (\overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ y\ \texttt{"main"}\ \texttt{"sqrt"}))\ \texttt{"sqrt"}\ \texttt{"main"})\ 4)$$

For the next few steps, we show the reductions of $\overline{wrap}$'s argument before the reduction of $\overline{wrap}$, for clarity.

$$\longrightarrow \overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ (sqrt\ (wrap\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ 4\ \texttt{"main"}\ \texttt{"sqrt"}))\ \texttt{"sqrt"}\ \texttt{"main"}$$

$$\longrightarrow^{*} \overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ (sqrt\ (\mathbf{if}\ ((\lambda\, x.x \geq 0)\ 4)\ \mathbf{then}\ 4\ \mathbf{else}\ blame(\texttt{"main"})))\ \texttt{"sqrt"}\ \texttt{"main"}$$

$$\longrightarrow^{*} \overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ (sqrt\ 4)\ \texttt{"sqrt"}\ \texttt{"main"}$$

$$\longrightarrow^{*} \overline{wrap}\ (\mathbf{contract}(\lambda\, x.x \geq 0))\ 2\ \texttt{"sqrt"}\ \texttt{"main"}$$

$$\longrightarrow^{*} \mathbf{if}\ (\lambda\, x.x \geq 0)\ 2\ \mathbf{then}\ 2\ \mathbf{else}\ blame(\texttt{"sqrt"})$$

$$\longrightarrow^{*} 2$$

Figure 10. Reducing *sqrt* in λ^{CON} and with *wrap*

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

5 Contract Implementation

To implement λ^{CON}, we must compile away obligation expressions. The key to the compilation is the wrapper function in figure 9. The wrapper function is defined in the calculus (the **let** expression is short-hand for inline applications of λ-expressions, and is used for clarity). It accepts a contract, a value to test, and two strings. These strings correspond to the variables in the superscripts. We write $\overline{wrap}$ as a meta-variable to stand for the program text in figure 9, *not* a program variable.

Compiling the obligations is merely a matter of replacing an obligation expression with an application of $\overline{wrap}$. 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 ($\mathcal{C}$) that maps from programs to programs. It replaces each obligation expression with the corresponding application of $\overline{wrap}$. The formal definition is given in the accompanying technical report [6].

The function $\overline{wrap}$ 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 the test fails. The second case of $\overline{wrap}$ deals with function contracts. It builds a wrapper function that tests the original function's argument and its result by recursive calls to $\overline{wrap}$. Textually, the first recursive call to $\overline{wrap}$ 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 $\overline{wrap}$ corresponds to the pre-condition checking. It applies the domain portion of the contract to the argument of the wrapper function. This call to $\overline{wrap}$ has the positive and negative blame positions reversed as befits the domain checking for a function.

The right-hand side of figure 10 shows how the compiled version of the *sqrt* program reduces. It begins with one call to $\overline{wrap}$ from the one obligation expression in the original program. The first reduction applies $\overline{wrap}$. Since the contract in this case is a function contract, $\overline{wrap}$ takes the second case in its definition and returns a λ expression. Next, the λ expression is applied to 4. 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 λ^{CON} (as shown on the left side of figure 10). The next reduction step is another call to $\overline{wrap}$, in the argument to *sqrt*. This contract is flat, so the first case in the definition of $\overline{wrap}$ 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 bad value to *sqrt*. The test passes, however, and the **if** expression returns 4 in the next reduction step.

$$
\mathcal{E}(p) \;=\; \begin{cases} <\!fn\!> & \text{if } \mathcal{C}(I(p)) \longrightarrow^{*} \lambda\, x.\, e \\ V_p & \text{if } \mathcal{C}(I(p)) \longrightarrow^{*} V_p \text{ and } V_p \neq \lambda\, x.\, e \\ error(x) & \text{if } \mathcal{C}(I(p)) \longrightarrow^{*} error(x) \end{cases}
$$

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

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

Figure 11. Evaluator Functions

After that, *sqrt* returns 2. Now we arrive at the final call to $\overline{wrap}$. As before, the contract is a flat predicate, so $\overline{wrap}$ 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.

6 Correctness

DEFINITION 6.1 DIVERGENCE. *A program p diverges under $\longrightarrow$ if for any p_1 such that $p \longrightarrow^{*} p_1$, there exists a p_2 such that $p_1 \longrightarrow p_2$.*

Although the definition of divergence refers only to $\longrightarrow$, we use it for each of the reduction relations.

The following type soundness theorem for λ^{CON} is standard [34].

THEOREM 6.2 (TYPE SOUNDNESS FOR λ^{CON}). *For any program, p, such that*

$\emptyset \vdash p : \langle\ t \cdots \rangle$

according to the type judgments in figure 7, exactly one of the following holds:

- $p \longrightarrow^{*} V_p : \langle\ t \cdots \rangle$
- $p \longrightarrow^{*}$ error(x)*, where x is a* **val rec** *defined variable in p, /, hd, tl, pred dom, or rng, or*
- *p diverges under $\longrightarrow$.*

PROOF. Combine the preservation and progress lemmas for λ^{CON}. □

LEMMA 6.3 (PRESERVATION FOR λ^{CON}). *If $\emptyset \vdash p : \langle\ t \cdots \rangle$ and $p \longrightarrow p'$ then $\emptyset \vdash p' : \langle\ t \cdots \rangle$.*

LEMMA 6.4 (PROGRESS FOR λ^{CON}). *If $\emptyset \vdash p : \langle\ t \cdots \rangle$ then either $p = V_p$, or $p \longrightarrow p'$, for some p'.*

The remainder of this section formulates and proves a theorem that relates the evaluation of programs in the instrumented semantics from section 4 and the contract compiled programs from section 5.

To relate these two semantics, we introduce a new semantics and show how it bridges the gap between them. The new semantics is an extension of the semantics given in figures 5 and 6. In addition to those expressions it contains obligation expressions, evaluation contexts, and $\xrightarrow{flat}$ reduction from figure 8 (but not the new values or the $\xrightarrow{hoc}$ reduction in figure 8), and the $\xrightarrow{wrap}$ reduction:

$$
\begin{array}{l} D[(\lambda\, x.\, e)^{(V_1 \longmapsto V_2),p,n}] \xrightarrow{wrap} \\ D[\lambda\, y.\, ((\lambda\, x.\, e)\ y^{V_1,n,p})^{V_2,p,n}] \end{array}
$$

where y is not free in e.

DEFINITION 6.5 (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 evaluator functions (shown in figure 11) are defined on programs p such that p **ok** *and $\Gamma \vdash p : \langle\ t \cdots \rangle$. As a short-hand notation, we write that a program value is equal to a value $V_p = V$ when the main expression of the program V_p is equal to V.*

LEMMA 6.6. *The evaluators are partial functions.*

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 (partial) functions. □

THEOREM 6.7 (COMPILER CORRECTNESS).

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

PROOF. Combine lemma 6.8 with lemma 6.9. □

LEMMA 6.8. $\mathcal{E} = \mathcal{E}_{fw}$

PROOF SKETCH. This proof is a straightforward examination of the evaluation sequences of $\mathcal{E}$ and $\mathcal{E}_{fw}$. Each reduction of an application of $\overline{wrap}$ corresponds directly to either a $\xrightarrow{flat}$ or a $\xrightarrow{wrap}$ reduction and otherwise the evaluators proceed in lock-step.

The full proof is given in an accompanying technical report [6]. □

LEMMA 6.9. $\mathcal{E}_{fw} = \mathcal{E}_{fh}$

PROOF SKETCH. This proof establishes a simulation between $\mathcal{E}_{fh}$ and $\mathcal{E}_{fw}$. The simulation is preserved by each reduction step and it relates values to themselves and errors to themselves.

The full proof is given in an accompanying technical report [6]. □

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 12.

dependent contract expressions

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

dependent contract type rule

$$\frac{\Gamma \vdash e_1 : t_1\ \textbf{contract} \qquad \Gamma \vdash e_2 : t_1 \longrightarrow (t_2\ \textbf{contract})}{\Gamma \vdash e_1 \overset{d}{\longmapsto} e_2 : (t_1 \longrightarrow t_2)\ \textbf{contract}}$$

dependent contract evaluation contexts

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

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 12. Dependent Function Contracts for λ^{CON}

The evaluation contexts given in figure 8 dictate that an obligation's superscript is reduced to a value before its base expression. In particular, this order of evaluation means that the superscripted application resulting from the dependent contract reduction in figure 12 is reduced before the base expression. Therefore, the procedure in the dependent contract can examine the state (of the world) before the function proper is applied. This order of evaluation is critical for the callback examples from section 2.5.

8 Tail Recursion

Since the contract compiler described in section 5 checks post-conditions, it does not preserve tail recursion [2, 30] 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:

val rec *gt0* $=$ **contract**($\lambda\ x.x \geq 0$)
val rec *f* : (*gt0* $\longmapsto$ *gt0*) $\longmapsto$ *gt0*
    $= \lambda\ g.\ g$ 3
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 the semantics of contract checking.[5]

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. For instance, imagine that the argument to *f* was a locally defined recursive function. The recursive calls would be tail-calls, since they would not be associated with any top-level variable, and thus no contract would be enforced.

Contracts are most effective at module boundaries, where they serve the programmer by improving the opportunities for modular reasoning. That is, with well-written contracts, a programmer can study a single module in isolation when adding functionality or fixing defects. In addition, if the programmer changes a contract, the changed contract immediately indicates which other source files must change.

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

Since 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 key interfaces in DrScheme has had no noticeable effect on its performance. Removing the tail-call optimization entirely, however, would render DrScheme useless.

Serrano presents further evidence for this conjecture about tail recursion. His compiler does not preserve tail recursion for any cross-module procedure call — not just those with contracts. Still, he has not found this to be a problem in practice [29, section 3.4.1].

9 Conclusion

Higher-order, typed programming language implementations [1, 12, 15, 19, 33] 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 these languages.

With the advent of module languages that support type abstraction [13, 18, 24], programmers are empowered to enforce their own abstractions at the type level. These abstractions have the same expressive power that the language designer used when specifying the language's primitives. The dynamic part of the invariant enforcement, 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, as discussed in section 2, it is not always possible for the programmer to insert these checks manually because the call sites may be in unavailable modules.

This paper presents the first assertion-based contract checker for languages with higher-order functions. Our contract checker enables programmers to refine the type-specifications of their abstractions with additional, dynamically enforced invariants. We illustrate the complexities of higher-order contract checking with a series of examples chosen from DrScheme's code-base. These examples serve two purposes. First, they illustrate the subtleties of contract checking for languages with higher-order functions. Second, they demonstrate that current static checking techniques are not expressive enough to support the contracts underlying DrScheme.

We believe that experience with assertions will reveal which contracts have the biggest impact on software quality. We hope that this information, in turn, helps focus type-system research in practical directions.

Acknowledgments

Thanks to Thomas Herchenröder, Michael Vanier, and the anonymous ICFP reviews for their comments on this paper.

We would like to send a special thanks to ICFP reviewer #3, whose careful analysis and insightful comments on this paper have renewed our faith in the conference reviewing process.

References

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

[2] 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.

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

[4] 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.

[5] 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*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.

[6] Findler, R. B. and M. Felleisen. Contracts for higher-order functions. Technical Report NU-CCS-02-05, Northeastern University, 2002.

[7] 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.

[8] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. http://www.plt-scheme.org/software/mzscheme/.

[9] Flatt, M. Composable and compilable macros: You want it *when?* In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2002.

[10] 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.

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

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

[13] Harper, R. and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM Conference Principles of Programming Languages*, pages 123–137, Janurary 1994.

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

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

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

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

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

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

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

[21] 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.

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

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

[24] Mitchell, J. C. and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

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

[26] 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.

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

[28] Serrano, M. *Bigloo: A practical Scheme compiler*, 1992–2002.

[29] Serrano, M. Bee: an integrated development environment for the Scheme programming language. *Journal of Functional Programming*, 10(2):1–43, May 2000.

[30] 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.

[31] Switzer, R. *Eiffel: An Introduction*. Prentice Hall, 1993.

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

[33] The GHC Team. *The Glasgow Haskell Compiler User's Guide*, 1999.

[34] 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.