Contracts as Pairs of Projections

Robert Bruce Findler[1] and Matthias Blume[2]

[1] University of Chicago
[2] Toyota Technological Institute at Chicago

Abstract. Assertion-based contracts provide a powerful mechanism for stating invariants at module boundaries and for enforcing them uniformly. In 2002, Findler and Felleisen showed how to add contracts to higher-order functional languages, allowing programmers to assert invariants about functions as values. Following up in 2004, Blume and McAllester provided a quotient model for contracts. Roughly speaking, their model equates a contract with the set of values that cannot violate the contract. Their studies raised interesting questions about the nature of contracts and, in particular, the nature of the `any` contract.

In this paper, we develop a model for software contracts that follows Dana Scott's program by interpreting contracts as projections. The model has already improved our implementation of contracts. We also demonstrate how it increases our understanding of contract-oriented programming and design. In particular, our work provides a definitive answer to the questions raised by Blume and McAllester's work. The key insight from our model that resolves those questions is that a contract that puts no obligation on either party is not the same as the most permissive contract for just one of the parties.

1 A Tour of Contracts

Assertion-based contracts play an important role in the construction of robust software. They give programmers a technique to express program invariants in a familiar notation with familiar semantics. Contracts are expressed as program expressions of type boolean. When the expression's value is true, the contract holds and the program continues. When the expression's value is false, the contract fails, the contract checker aborts the program, and hopefully, it identifies the violation and the violator. Identifying the faulty part of the system helps programmers narrow down the cause of the violation and, in a component-oriented programming setting, exposes culpable component producers.

The idea of software contracts dates back to the 1970s [31]. In the 1980s, Meyer developed an entire philosophy of software design based on contracts, embodied in his object-oriented programming language Eiffel [30]. Nowadays, contracts are available in one form or another for many programming languages (*e.g.*, C [37], C++ [33], C# [29], Java [1, 4, 7, 20, 22, 24, 26], Perl [5], Python [32], Scheme [14, 35], and Smalltalk [3]). Contracts are currently the third most requested addition to Java.[1] In C code, assert statements are particularly popular, even though they do not have enough information to properly assign blame and thus are a degenerate form of contracts. In fact, 60% of

[1] `http://bugs.sun.com/bugdatabase/top25_rfes.do` as of 1/20/2006

M. Hagiya and P. Wadler (Eds.): FLOPS 2006, LNCS 3945, pp. 226–241, 2006.

the C and C++ entries to the 2005 ICFP programming contest [10] used assertions, despite the fact that the software was produced for only a single run and was ignored afterwards.

The "hello world" program of contract research is:

```
float sqrt(float x) { ... }
// @pre{ x >= 0 }
// @post{ @ret >= 0 && abs(x − @ret * @ret) <= 0.01 }
```

The pre-condition for `sqrt` indicates that it only receives positive numbers, and its post-condition indicates that its result is positive and within `0.01` of the square root of its input. If the pre-condition contract is violated, the blame is assigned to the caller of `sqrt`, but if the post-condition is violated, the blame is assigned to `sqrt` itself.

Until relatively recently, functional languages have not been able to benefit from contract checking, and with what might seem to be a good reason. Because functional languages permit functions to be used as values, contract checking must cope with assertions on the behavior of functions, *i.e.*, objects with infinite behavior. For example, this contract restricts `f`'s argument to functions on even integers:

```
let f(g : (int{even} → int{even})) : int = ... g(2) ...
```

But what can it mean for a function to only accept functions on even numbers? According to Rice's theorem, this property is not decidable.[2]

Rather than try to check a function's behavior when we first encounter it, we can — in keeping with the spirit of dynamically enforced contracts — wait until each function is called with or returns simple values and only at that point check to see if the values match the contract.[3]

Once contract checking is delayed, blame assignment becomes subtle. In general, the blame for a contract violation lies with the party supplying the value at the point where the contract violation occurs. In a first-order setting, the caller first supplies a value to a function and it responds with another value. Thus the caller is responsible for the entire contract on the input and the function is responsible for entire contract on the result. In the higher-order function world, however, this reasoning is too simplistic.

Consider the situation where `f` (as above) is called with the function `λx.x+1`, and `f`, as shown, calls its argument with the number `2`. At this point, a contract violation occurs, because `3` is produced, but `3` is not an even integer. Clearly, the blame for the contract violation cannot lie with `f`, because `f` called its argument with a valid input. Instead, the blame for the violation must lie with `f`'s caller, because it did not provide a suitable function. In a similar fashion, if `f` had supplied `3` to its argument, `f` would be to blame.

To generalize from the first-order setting, we need to observe that all of the negative positions in the contract (those positions that occur to the left of an odd number of

[2] Object-oriented programming languages share this problem with higher-order functional languages. In particular, it is impossible to check whether a contract concerning behavioral subtyping holds until the classes are instantiated and the relevant methods are invoked [13, 16]. We focus here on the functional setting because it is simpler than the object-oriented one.

[3] And thus, in answer to the age-old question, no: the tree does not make a sound if no one is there to hear it fall. In fact, it didn't even fall until someone sees it on the ground.

arrows) are points at which the context is supplying values and therefore the context must be blamed for any violations of those parts of the contract. Similarly, all of the positive positions in the contract (those that occur to the left of an even number of arrows) are points where the function supplies values to its context and thus the function must be blamed for any violations of those parts of the contract. In our running example, `f` is responsible for the inputs to the function it receives, and `f`'s caller is responsible for the results of that function.

In the first order setting, the negative and positive positions of the contract match the pre- and post-conditions for a function, making traditional pre- and post-condition checking a natural specialization of higher-order contract checking.

The remainder of this paper explores models of higher-order contracts. The next section introduces the formal setting for the paper. Section 3 shows how our original contract checker is in fact a disguised version of projections. Section 4 introduces projections and discusses orderings on projections. Section 5 relates projections to Blume and McAllester's model of contracts. Equipped with this background, section 6 revisits Blume and McAllester's motivating example, and section 7 concludes.

2 Modeling Scheme and Contracts

For the rest of this paper, we focus on an idealized, pure version of Scheme [17, 21, 27], and source programs that contain a single contract between two parties in the program. The syntax and semantics for this language is given in figure 1. A program consists of a series of definitions followed by a single expression (ellipses in the figure indicate (zero or more) repeated elements of whatever precedes the ellipsis). Definitions associate variables with expressions and expressions consist of λ expressions, applications, variables, symbolic constants (written as a single quote followed by a variable name), integers, booleans, **`if`** expressions, primitives for `cons` pairs, the three primitive predicates, `procedure?` `integer?`, and `pair?`, and an expression to assign blame.

The operational semantics is defined by a context-sensitive rewriting system in the spirit of Felleisen and Hieb [9]. Contexts are non-terminals with capital letters (`P`, `D`, `E`) and allow evaluation in definitions, from left-to-right in applications, in the test position of **`if`** expressions, and in **`blame`** expressions. The evaluation rules are standard: β_v for function application, the predicates `procedure?`, `integer?`, and `pair?` recognize λs, integers, and `cons` pairs respectively, `car` and `cdr` extract the pieces of a `cons` pair, and **`if`** chooses between its second and third arguments (unlike in standard Scheme, our **`if`** requires the test to be a boolean). Variables bound by **`define`** are replaced with their values, and finally **`blame`** aborts the program and identifies its argument as faulty.

The syntactic shorthands allow us to write examples later in the paper in a clear manner, but without cluttering the language and it's semantics unduly. The composition operator, in particular, is defined to evaluate its arguments before performing the composition in order to match a standard functional definition, to avoid variable capture and associated machinery [8, 23], and to make later computations simpler.

Contracts belong on module boundaries, mediating the interaction between coherent parts of a program. Rather than build a proper module system into our calculus, however, we divide the program into two parts: an arbitrary context (not just an eval-

syntax

```
p = d ... e
d = (define x e)
e = (λ (x ...) e) | (e e ...) | x | 'x | i | #t | #f | (if e e e)
  | cons | car | cdr | procedure? | integer? | pair? | (blame e)

P = dv ... D d ... e | dv ... E
D = (define x E)
E = (v ... E e ...) | (if E e e) | (blame E) | □

dv = (define x v)
v = (λ (x ...) e) | (cons v v) | 'x | i | #t | #f
  | cons | car | cdr | procedure? | integer? | pair?
i = integers
x = variables
```

operational semantics

```
P[((λ (x ...) e) v ...)] ⟶ P[{x/v ...}e] ;; #x = #v
P[(integer? i)] ⟶ P[#t]
P[(integer? v)] ⟶ P[#f] ;; v not an integer
P[(procedure? (λ (x ..) e))] ⟶ P[#t]
P[(procedure? v)] ⟶ P[#f] ;; v not a λ expression
P[(pair? (cons v1 v2))] ⟶ P[#t]
P[(pair? v)] ⟶ P[#f] ;; v not a cons pair
P[(car (cons v1 v2))] ⟶ P[v1]
P[(cdr (cons v1 v2))] ⟶ P[v2]
P[(if #t e1 e2)] ⟶ P[e1]
P[(if #f e1 e2)] ⟶ P[e2]
P[x] ⟶ P[v] ;; where (define x v) is in P
P[(blame 'x)] ⟶ x violated the contract
```

syntactic shorthands

```
(define (f x ...) e) = (define f (λ (x ...) e))
(let ([x e1] ...) e2) = ((λ (x ...) e2) e1 ...)
(cond [e1 e2] [e3 e4] ...) = (if e1 e2 (cond [e3 e4] ...))
(cond) = #f
(e1 ∘ e2) = (let ([x1 e1][x2 e2]) (λ (y) (x1 (x2 y))))
```

Fig. 1. Syntax and semantics for a core Scheme

uation context) and a closed expression in the hole of the context, with a contract at the boundary. We call the context the client and the expression the server; the contract governs the interaction between the client and the server. Separating the program in this manner is, in some sense, the simplest possible model of a module language. Although it does not capture the rich module systems available today, it does provide us with a simple setting in which to effectively study contracts and contract checking.

As examples, consider the following clients, contracts, and servers:

Client	Contract	Server
`(□ 2)`	`odd → odd`	`(λ (y) y)`
`(□ 3)`	`odd → odd`	`(λ (y) (− (* y y) y))`
`(□ (λ (x) (+ x 2)))`	`(odd → odd) → even`	`(λ (f) (f 1))`

The first contract says that the server must be a function that produces odd numbers and that the client must supply odd numbers, but when plugging the server expression into the hole (□) in the client context, the client calls the server function with `2`, so it is blamed for the contract violation. In the second line, the client correctly supplies an odd number, but the server produces an even number, and so must be blamed. In the third line, the client supplies a function on odd numbers to the server. The server applies the function to `1`, obeying the contract. The server then receives `3` from the client, discharging the client's obligation to produce odd numbers, but the server returns that `3`, which is not an even number and thus violates the contract; this time, the server broke the contract and is blamed for the violation.

3 Re-functionalizing the Contract Checker

A specification of contracts for a language with atomic values and single-argument functions boils down to three functions:

```
 flat : (α → boolean) → contract α
   ho : contract α × contract β → contract (α → β)
guard : contract α × α × symbol × symbol → α
```

The `flat` and `ho` functions are combinators that build contracts. The function `flat` consumes a predicate and builds a contract that tests the predicate. Usually, `flat` is applied to predicates on flat types, like numbers or booleans. In languages that have richer function types, *e.g.*, multi-arity functions or keyword arguments, `flat` can be used to construct contracts that test flat properties of functions, such as the arity or which keywords the function accepts. The function `ho` builds a contract for a function, given a contract for the domain and a contract for the the range. As an example, `(ho (flat odd?) (flat odd?))` is the contract from the first example in section 2, given a suitable definition of `odd?`. To enforce a contract, `guard` is placed into the hole in the client context, around the server expression. Its first argument is the contract (built using `flat` and `ho`). Its second argument is the server, and its last two arguments name the server and the client, and are used to assign blame. When fully assembled, the first example from section 2 becomes:

```
((guard (ho (flat odd?) (flat odd?))
        (λ (y) y)
        'server 'client)
 2)
```

In earlier work [14], we provided the first implementation of that interface. In that implementation, the contract construction functions were just record constructors and the interesting code was in the `guard` function, as shown in figure 2. The $\texttt{flat}_1$ and $\texttt{ho}_1$ functions collect their arguments. The $\texttt{guard}_1$ function is defined in cases based on the

```
;; data Contract₁ α where
;;   Flat :: (α → Bool) → Contract α
;;   Ho :: Contract α → Contract β → Contract (α → β)

(define (flat₁ p) p)
(define (ho₁ dom rng) (cons dom rng))

(define (guard₁ ctc val pos neg)
  (cond
    [(procedure? ctc)
     (if (ctc val) val (blame pos)))]
    [(pair? ctc)
     (let ([dom (car ctc)]
           [rng (cdr ctc)])
       (if (procedure? val)
           (λ (x)
             (guard₁ rng
                     (val (guard₁ dom x neg pos))
                     pos
                     neg))
           (blame pos)))]))
```

Fig. 2. Original contract library implementation

structure of the contract. If the contract is a flat contract, the corresponding predicate is applied and either blame is assigned immediately, or the value is just returned. If the contract is a higher-order function contract, the value is tested to make sure it is a procedure; if so, another function is constructed that will, when applied, ensure that the inputs and outputs of the function behave according to the domain and range contracts. The last two arguments to `guard₁` are reversed in the recursive call for the domain contract, but remain in the same order in the recursive call for the range contract. This reversal ensures proper blame assignment for the negative and positive positions of the contract.

Without types, we can represent a higher-order function contract as a pair of contracts and a flat contract as the corresponding predicate, but written in this manner, the program would not type-check in SML or Haskell. It does type-check, however, if we use the generalized abstract datatype [19, 42] `Contract₁`, shown as a comment in figure 2.

The `Contract₁` datatype constructors can be viewed as two defunctionalized functions [36], and `guard₁` as the defunctionalized version of apply.[4] To re-functionalize the program, we can move the code in the first **cond** clause of `guard₁` to a function in the body of the flat contract combinator, move the code from the second **cond** clause to a function in the body of the higher-order contract combinator, and replace the body of `guard₁` by a function application. The new type for contracts is thus a function that accepts all of the arguments that `guard₁` accepts (except the contract itself), and produces the same result that `guard₁` produces. If we clean up that implementation a little bit by currying contracts and then lifting out partial applications in the body of `ho`, we get the code in figure 3.

[4] Yang [43] and Danvy & Nielsen [6] have also explored similar transformations, in more detail.

```
;; type Contract₂ α = symbol × symbol → α → α

(define (flat₂ pred?)
  (λ (pos neg)
    (λ (val)
      (if (pred? val) val (blame pos)))))

(define (ho₂ dom rng)
  (λ (pos neg)
    (let ([dom-p (dom neg pos)]
          [rng-p (rng pos neg)])
      (λ (val)
        (if (procedure? val)
            (λ (x) (rng-p (val (dom-p x))))
            (blame pos))))))

(define (guard₂ ctc val pos neg) ((ctc pos neg) val))
```

Fig. 3. Re-functionalized, cleaned up contract implementation

These two transformations lead to a significantly improved implementation, for two reasons:

– The new implementation is more efficient. PLT Scheme comes with a full featured contract checking library that includes over 60 contract combinators and several different ways to apply contracts to values [35, Chapter 13]. We changed PLT Scheme's contract library from an implementation based on the code in figure 2 to one based on the code in figure 3 and checking a simple higher-order contract in a tight loop runs three times faster than it did before the change. Of course, PLT Scheme does not contain a sophisticated compiler, and the performance improvement for such a implementations is likely to be less dramatic. For example, in ghc-6.4.1 [40] on a 1.25 GHz PowerPC G4, the figure 3 version of a toy contract library is 25% faster than a version similar to the one in figure 2, but written with pattern matching.
– The new implementation is easier to extend. Adding contracts for compound data like pairs and lists is simply a matter of writing additional combinators. For example, a combinator for immutable cons pairs can be defined without changing the existing code:

```
;; pair/c : contract α × contract β → contract (α × β)
(define (pair/c lhs rhs)
  (λ (pos neg)
    (let ([lhs-p (lhs pos neg)]
          [rhs-p (rhs pos neg)])
      (λ (x) (if (pair? x)
                 (cons (lhs-p (car x)) (rhs-p (cdr x)))
                 (blame pos))))))
```

4 Contracts as Pairs of Error Projections

Even more striking than the implementation improvements is that the text of the body of the `ho2` contract combinator is identical to Scott's function space retract and the text of the body of the `pair/c` contract combinator is identical to his retract for pairs [39]. The correspondence between our contracts and Scott projections is not mere syntactic coincidence; there is a semantic connection and the rest of this paper explores that connection in depth.

Scott defined projections (p) as functions (technically, elements in the domain $\mathbf{P}\omega$) that have two properties:

1. $p = p \circ p$
2. $p \sqsubseteq 1$

The first, called the retract property, states that projections are idempotent on their range. The second says that the result of a projection contains no more information than its input. The equations also make intuitive sense for contracts. The first means that it suffices to apply a contract once; the second means that a contract cannot add behavior to a value. The second rule is not quite right for a contract checker, however, because the contract must be free to identify erroneous programs by signaling errors. Instead, we insist on a slightly different property, namely that the only behavior that a contract adds are such errors, and otherwise the contract leaves its input untouched. We call such functions error projections. The `ho` contract combinator always produces error projections from error projections and `flat` produces error projections for first-order inputs and produces error projections when its predicate does not explore the higher-order behavior of its argument (as we showed in earlier work [12]).

Retracts have a natural ordering, as defined by Scott [39]

$$a \propto b \text{ if and only if } a = a \circ b$$

When viewed as an ordering on contracts, it relates two retracts a and b if a signals a contract violation at least as often as b, but perhaps more. Intuitively, it captures the strength of the contract. A contract that ignores its argument and always signals an error is the smallest contract (*i.e.*, it likes the fewest values), and the identity function is the largest contract (*i.e.*, it likes the most values).

Given this ordering and the `ho2` contract combinator, it is natural to ask if the ordering is contra-variant in the domain and co-variant in the range of `ho2`, analogous to conventional type systems. Disappointingly, as noted by Scott, it is co-variant in the domain.

Theorem 1. *(Scott [39]) For any retracts,* $\mathtt{d}_1$*,* $\mathtt{d}_2$*, and* $\mathtt{r}$*, if* $\mathtt{d}_1 \propto \mathtt{d}_2$*, then* $(\mathtt{ho}_2\ \mathtt{d}_1\ \mathtt{r}) \propto (\mathtt{ho}_2\ \mathtt{d}_2\ \mathtt{r})$.

Proof. Assume that $\mathtt{d}_1 \propto \mathtt{d}_2$, and consider the composition of $(\mathtt{ho}_2\ \mathtt{d}_1\ \mathtt{r})$ and $(\mathtt{ho}_2\ \mathtt{d}_2\ \mathtt{r})$

```
   (ho₂ d₁ r) ∘ (ho₂ d₂ r)
=  (λ (f) (λ (x) (r (f (d₁ x))))) ∘          ;; definition of ho₂
   (λ (f) (λ (x) (r (f (d₂ x)))))
=  (λ (f) ((λ (f) (λ (x) (r (f (d₁ x)))))    ;; definition of
          ((λ (f) (λ (x) (r (f (d₂ x)))))    ;;    composition & let,
           f)))                              ;;    and βv
=  (λ (f) ((λ (f) (λ (x) (r (f (d₁ x)))))    ;; βv
          (λ (x) (r (f (d₂ x))))))
=  λ (f) (λ (x) (r ((λ (x) (r (f (d₂ x))))   ;; βv
                    (d₁ x))))
=  (λ (f) (λ (x) (r (r (f (d₂ (d₁ x)))))))   ;; βω [38]
=  (λ (f)                                    ;; apply retract law,
     (λ (x) (r (f (d₂ (d₁ x))))))            ;;    to eliminate one r
=  (λ (f) (λ (x) (r (f (d₁ x)))))            ;; by assumption & lemma (below)
=  (ho₂ d₁ r)                                ;; definition of ho₂
```

The steps above use the lemma that for retracts a and b, $a = a \circ b$ implies $a = b \circ a$. □

Thus, because functions are naturally contra-variant in their arguments, this ordering fails to properly capture the ordinary reasoning rules about functions. Inspecting the analogy between contracts and error projections, we see that the Scott ordering ignores the blame associated with contracts. To cope with blame, we must first separate each contract into two projections: one that assigns blame to the client and one that assigns blame to the server, and then we can compare the projections separately. A violation of the first projection in the pair indicates the server is to blame and a violation of the second indicates the client is to blame.

Concretely, we represent contracts as pairs of error projections that are parameterized over the guilty party. We assume, however, that the parameterized projection does not dispatch on the symbol, and when it does assign blame, it always assigns blame to the symbol is received as an argument. Figure 4 shows the new implementation of the contract combinators.

As before, the sense of the blame is reversed for the domain side of a function contract. This reversal is captured in this version of the combinators by using the client's part of the domain (`ac`) in the server part of `ho`'s result (the `car` position) and using the server's part of the domain (`as`) in the client part in the result (the `cdr` position).

To show that the new higher-order contract combinator checks the contracts in the same manner as the one in figure 3, we can construct suitable inputs for both combinators from a single set of error projections, and show that they produce the same higher-order projection.

Theorem 2. *For any values* `a, b, c, d : symbol` $\rightarrow \alpha \rightarrow \alpha$

```
(ho₂ (λ (pos neg) ((a pos) ∘ (b neg)))
     (λ (pos neg) ((c pos) ∘ (d neg))))
=
(let ([pr (ho₃ (cons a b)
               (cons c d))])
  (λ (pos neg)
    ((car pr) pos) ∘ ((cdr pr) neg)))
```

```
;; type Contract3 α = (symbol → α → α) × (symbol → α → α)

(define (flat3 f)
  (cons (λ (s) (λ (x) (if (f x) (blame s) x)))
        (λ (s) (λ (x) x))))

(define (ho3 a b)
  (cons (λ (s)
          (let ([ac ((cdr a) s)]
                [bs ((car b) s)])
            (λ (val)
              (if (procedure? val)
                  (λ (x) (bs (val (ac x))))
                  (blame s)))))
        (λ (s)
          (let ([bc ((cdr b) s)]
                [as ((car a) s)])
            (λ (val)
              (if (procedure? val)
                  (λ (x) (bc (val (as x))))
                  val))))))

(define (guard3 ctc val pos neg)
  (let ([server-proj ((car ctc) pos)]
        [client-proj ((cdr ctc) neg)])
    (client-proj (server-proj val))))
```

Fig. 4. Contract combinators for contracts as pairs of projections

Proof (sketch). The proof is an algebraic manipulation in Sabry and Felleisen's equational theory $\lambda\beta_v X$ [34, 38] (without η_v) extended with δ rules for **`if`** [28]. For details, see the accompanying tech report [11]. □

To define a blame-sensitive ordering, we must take into account the difference between contracts that blame the client and contracts that blame the server. In particular, assigning blame more often to the client means that *more* servers are allowed, whereas assigning blame less often to the client means *fewer* servers are allowed.

Definition 1 ($\propto\!\!<\!\!<$)

$$\texttt{(cons } a_s\, a_c\texttt{)} \propto\!\!<\!\!< \texttt{(cons } b_s\, b_c\texttt{)}$$
$$\textit{if and only if}$$
$$\texttt{(}a_s\ \texttt{s)} \propto\!< \texttt{(}b_s\ \texttt{s)} \quad \textit{and} \quad \texttt{(}b_c\ \texttt{s)} \propto\!< \texttt{(}a_c\ \texttt{s)} \ \textit{for any symbol}\ \texttt{s}.$$

Theorem 3. *The relation* $\propto\!\!<\!\!<$ *is a partial order.*

Proof. Follows directly from the fact that $\propto\!<$ is a partial order (Scott [39]). □

Theorem 4. *For any error projections,* $\mathtt{d}_1$, $\mathtt{d}_2$, *and* `r`, *and symbol* `s`,

$$\mathtt{d}_1 \propto\!\!< \mathtt{d}_2 \quad \textit{implies} \quad (\mathtt{ho}_3\ \mathtt{d}_2\ \mathtt{r}) \propto\!\!< (\mathtt{ho}_3\ \mathtt{d}_1\ \mathtt{r}).$$

Proof (sketch). This proof is an algebraic manipulation using the equations in the proof of theorem 2 and C_{lift} [9] used for **`blame`**, plus the lemma that, for any two retracts a and b, if $a = a \circ b$ then $a = b \circ a$. For details, see the accompanying tech report [11]. □

In short, a blame-sensitive ordering provides one that is naturally contra-variant in the domain of the functions.

5 Ordering Contracts in the Blume-McAllester Model

The quotient model of contracts proposed by Blume and McAllester [2] also leads to an ordering on contracts. This section revisits their model and connects the $\propto\!\!<$ ordering to the ordering in their model.

In Blume and McAllester's work, contracts (`c`) are either function contracts or predicates that never signal errors, diverge, or get stuck.[5]

$$\mathtt{c} = \mathtt{c} \rightarrow \mathtt{c} \mid (\lambda\ (\mathtt{x})\ \mathtt{e})$$

The meaning of each contract is a set of terms representing values that satisfy the contract. The values inhabiting higher-order function contracts are procedures that, when given an input in the domain contract, produce an output in the range contract or diverge. The values inhabiting flat contracts are the safe values that match the flat contract's predicate. Safe values are either first-order values, or functions that map safe arguments to safe results (or diverge). In other words, safe values can never be the source of an error.

Definition 2. *The set* ***Safe*** *is the largest subset of the set of values* `v` *such that each element of* ***Safe*** *is either:*

1. *an integer,* `#t`, `#f`, *or*
2. $(\lambda\ (\mathtt{x})\ \mathtt{e})$ *where, for each value* $\mathtt{v}_1$ *in* ***Safe****, either* $((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{v}_1) \longrightarrow^* \mathtt{v}_2$ *and* $\mathtt{v}_2$ *is in* ***Safe****, or* $((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{v}_1)$ *diverges.*

An expression `e` diverges if, for all $\mathtt{e}_2$ such that $\mathtt{e} \longrightarrow^* \mathtt{e}_2$, there exists an $\mathtt{e}_3$ such that $\mathtt{e}_2 \longrightarrow \mathtt{e}_3$. Blume and McAllester showed that definition 2 is well-formed [2].

Given **Safe**, we can formally define the meaning of contracts.

Definition 3. $[\![\cdot]\!] : \mathtt{c} \rightarrow \{\mathtt{v}\}$

$$[\![(\lambda\ (\mathtt{x})\ \mathtt{e})]\!] = \{\mathtt{v} \in \textbf{\textit{Safe}} \mid ((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{v}) \longrightarrow^* \mathtt{\#t}\}$$

$$[\![c_1 \rightarrow c_2]\!] = \left\{ (\lambda\ (\mathtt{x})\ \mathtt{e}) \;\middle|\; \begin{array}{l} \forall \mathtt{v}_1 \in [\![c_1]\!]. \\ \quad ((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{v}_1) \longrightarrow^* \mathtt{v}_2 \textit{ and } \mathtt{v}_2 \in [\![c_2]\!] \\ \quad \textit{or} \\ \quad ((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{v}_1) \textit{ diverges} \end{array} \right\}$$

[5] In their work, contracts are formulated differently, but these differences are minor. Their safe is `(λ (x) #t)`, their int is `(λ (x) (integer? x))`, and our `(λ (x) e)` is $\langle \underline{\text{safe}} \mid \lambda x.e \rangle$.

The subset ordering ($\subseteq$) on the sets of values produced by $[\![\cdot]\!]$ induces an ordering on contracts and we can ask how that ordering relates to $\propto\!\!<$. To do so, we first map Blume and McAllester's contracts to error projections, via $(\!|\cdot|\!)$.

Definition 4. $(\!|\cdot|\!)$ `:c` $\rightarrow$ `e`

$$(\!|\texttt{(}\lambda\texttt{ (x) e)}|\!) = \texttt{(flat}_3\texttt{ (}\lambda\texttt{ (x) e))}$$
$$(\!|c_1 \rightarrow c_2|\!) = \texttt{(ho}_3\texttt{ } (\!|c_1|\!) \; (\!|c_2|\!)\texttt{)}$$

We would like the two ordering relations to be the same but unfortunately $\subseteq$ relates slightly more contracts than $\propto\!\!<$. First we note that if the error projection ordering relates two contracts, so does the set model's ordering.

Theorem 5. *For any c, c':* $(\!|c|\!) \propto\!\!< (\!|c'|\!) \;\Rightarrow\; [\![c]\!] \subseteq [\![c']\!]$

Proof (sketch). The proof is a simultaneous induction on the structure of c and c', and is given in the accompanying technical report [11]. □

The reverse direction does not hold for every pair of contracts. Consider these two contracts in the Blume-McAllester model:

```
(λ (x) false) → (λ (x) false)        (λ (x) false) → (λ (x) true)
```

In both cases, the range contract is irrelevant, because the domain contract always rejects all values. Accordingly, they both map to the same set of values under $[\![\cdot]\!]$. The corresponding pairs of error projections, however,

```
(define p1 (ho3 (flat3 (λ (x) false)) (flat3 (λ (x) false))))
(define p2 (ho3 (flat3 (λ (x) false)) (flat3 (λ (x) true))))
```

are not the same and, in particular, $\texttt{p}_2 \propto\!\!< \texttt{p}_1$ does not hold.

Still, the two orders are related when we restrict higher-order function contracts in a minor way. In particular, every flat contract that appears as the domain position of a function contract must accept at least one value. In practice, this restriction is minor, because functions that always fail when applied are not generally useful. To express this restriction formally, we define a sub-language of `c`, called $\hat{\texttt{c}}$:

```
   ĉ = ne-c | (λ (x) e)
ne-c = ne-c → ĉ | non-empty-predicate
```

where `non-empty-predicate` stands for flat predicates that accept at least one value.

Theorem 6

1. *There exists* `c` *and* `c'` *such that* $[\![\texttt{c}]\!] \subseteq [\![\texttt{c}']\!] \;\Rightarrow\; (\!|\texttt{c}|\!) \not\propto\!\!< (\!|\texttt{c}'|\!)$
2. *For any* $\hat{\texttt{c}}$, $\hat{\texttt{c}}'$: $[\![\hat{\texttt{c}}]\!] \subseteq [\![\hat{\texttt{c}}']\!] \;\Rightarrow\; (\!|\hat{\texttt{c}}|\!) \propto\!\!< (\!|\hat{\texttt{c}}'|\!)$

Proof (sketch). The first part follows from the example above. The proof of the second part is a simultaneous induction on the structure of $\hat{\texttt{c}}$ and $\hat{\texttt{c}}'$, and is given in the accompanying technical report [11]. □

6 Revisiting the Blume-McAllester Example

Now that we have developed an ordering on contracts and can treat contracts as error projections, we can revisit Blume & McAllester's motivating example [2]:

Client	**Contract**	**Server**

`(let ([invert`
`        (λ (y) (/ 1 y))])`
`  ((□ invert) 0))`
	`(non-zero-num? → num?)`
	`→`
	`any`
		`(λ (x) x)`

According to the contract between the context and the expression, `invert` must not receive zero as input. But when we put the identity function into the hole of the context, `invert` is applied to `0`. So, someone must be blamed. The key question is whom?

There are two seemingly intuitive answers for this question. Here is the one that Blume & McAllester put forth (paraphrased):

> *The* `(λ (y) (/ 1 y))` *flows into the domain contract,* `non-zero-num? → num?` *and then back out into* `any`*. Clearly,* `non-zero-num? → num?` *should be a subcontract of* `any`*, because* `any` *accepts any value and thus is the highest contract in the subtyping ordering. Accordingly, we cannot blame* `(λ (x) x)`.

Here's the one that Findler & Felleisen saw, when they first looked at this expression:

> *The expression* `(λ (x) x)` *accepts a function with a requirement that it not be abused. It then lets that function flow into a context that may do anything (and thus promises nothing), because its contract is* `any`*. So,* `(λ (x) x)` *must be blamed for failing to protect its argument.*

These two intuitive explanations are clearly in conflict. Surprisingly, both have a correct interpretation in our model of contracts as projections, depending on the meaning of the word "any" and the corresponding choice of the `any` projection pair.

To see how, we can start by simplifying the program according to the definitions of the contract combinators, as shown in figure 5. The first expression shows the client, contract, and server combined into a single expression. The second expression shows how the domain contract is distributed to `invert` and the range contract is distributed to the result of the applying `(λ (x) x)` to `invert`. The inner `guard` expression corresponds to the domain part of the original contract, so the arguments to `guard` are reversed from their original senses, meaning that the client is responsible for results of `invert` and the server is responsible for the arguments to `invert`. The third expression shows how the inner `guard` is distributed into the body of `invert`. Again, the arguments to `guard` are reversed for the domain, leaving the `server` responsible for the value of `y`. At this point, we are left with the contract `any` applied to a procedure.

To support Blume & McAllester's answer, we must interpret `any` as the highest contract in the $\multimap\!\ll$ ordering,

```
(cons (λ (s) (λ (x) x))
      (λ (s) (λ (x) (blame s))))
```

With this interpretation of `any`, the client is immediately blamed, as they predict.

```
(((guard ⦇(non-zero-num → num) → any⦈ (λ (x) x) 'server 'client)
  invert)
 0)
= ((guard ⦇any⦈
          ((λ (x) x) (guard ⦇non-zero-num → num⦈
                            invert
                            'client 'server))
          'server 'client)
   0)
= ((guard ⦇any⦈
          (λ (y) (guard ⦇num⦈
                        (/ 1 (guard ⦇non-zero-num⦈ y 'server 'client))
                        'client 'server))
          'server 'client)
   0)
```

Fig. 5. Distributing the Contracts in the Blume-McAllester Example

To support Findler & Felleisen's answer, we must interpret `any` as the contract that never assigns blame,

```
(cons (λ (s) (λ (x) x))
      (λ (s) (λ (x) x)))
```

With this `any`, the outer `guard` in the last expression of figure 5 simply disappears. Thus, when the context supplies `0` to `invert` the latent guards assign blame to the server, as they predict.

Now that we have both projection pairs, we can ask which interpretation of `any` is more useful in practice. While such a judgment call is not supported by the model, it seems clear that the top of the ordering is a less useful contract, because it will always immediately abort the computation with a contract violation. The contract that never assigns blame, however, is useful because it allows us to build contracts that specify some properties, but leave others undetermined.

7 Conclusion

The Blume-McAllester contract example focuses our attention on an important lesson for contract programmers: the contract that never assigns blame is not the most permissive; the contract that always blames someone else is. Of course, finding partners that would agree to such a contract is a Phyrric victory, because it is impossible to achieve a useful goal with a contract that is always violated. As in real life, so too in programming: you've got to give a little to get a little.

Ever since their initial appearance in Scott's work, projections have enjoyed a wide use. Wadler and Hughes used them for strictness analysis [41], Launchbury used them for partial evaluation [25], and in our own work, projections have enabled us to build better models for contracts [12], to use contracts to connect nominal and structural type systems in a single language [15], and to interoperate between Java and Scheme [18]. We believe that this work is just the tip of the iceberg and intend to explore them further.

Acknowledgments. Thanks to Bob Harper for alerting us to the connection between contracts and retracts and to Matthias Felleisen and the FLOPS 2006 reviewers for their comments on this paper.

References

1. Bartetzko, D. Parallelität und Vererbung beim Programmieren mit Vertrag. Diplomarbeit, Universität Oldenburg, April 1999.
2. Blume, M. and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, to appear.
3. Carrillo-Castellon, M., J. Garcia-Molina, E. Pimentel and I. Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 7(9):23–28, 1996.
4. Cheon, Y. A runtime assertion checker for the Java Modelling Language. Technical Report 03-09, Iowa State University Computer Science Department, April 2003.
5. Conway, D. and C. G. Goebel. Class::Contract – design-by-contract OO in Perl. `http://search.cpan.org/~ggoebel/Class-Contract-1.14/`.
6. Danvy, O. and L. R. Nielsen. Defunctionalization at work. In *International Conference on Principles and Practice of Declarative Programming*, 2001.
7. Duncan, A. and U. Hölzle. Adding contracts to Java with handshake. Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998.
8. Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
9. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, pages 235–271, 1992.
10. Findler, Barzilay, Blume, Codik, Felleisen, Flatt, Huang, Matthews, McCarthy, Scott, Press, Rainey, Reppy, Riehl, Spiro, Tucker and Wick. The eighth annual ICFP programming contest. `http://icfpc.plt-scheme.org/`.
11. Findler, R. B. and M. Blume. Contracts as pairs of projections. Technical Report TR-2006-01, University of Chicago Computer Science Department, 2006. `http://www.cs.uchicago.edu/research/publications/techreports/TR-2006-01`.
12. Findler, R. B., M. Blume and M. Felleisen. An investigation of contracts as projections. Technical Report TR-2004-02, University of Chicago Computer Science Department, 2004.
13. Findler, R. B. and M. Felleisen. Contract soundness for object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.
14. Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2002.
15. Findler, R. B., M. Flatt and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.
16. Findler, R. B., M. Latendresse and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ACM Conference Foundations of Software Engineering*, 2001.
17. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. http://www.mzscheme.org/.
18. Gray, K. E., R. B. Findler and M. Flatt. Fine-grained interoperability through contracts and mirrors. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2005.
19. Jones, S. P., G. Washburn and S. Weirich. Wobbly types: Practical type inference for generalised algebraic dataypes. `http://www.cis.upenn.edu/~sweirich/publications.html`.

20. 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.
21. Kelsey, R., W. Clinger and J. Rees (Editors). Revised^5 report of the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
22. 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.
23. Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
24. Kramer, R. iContract: The Java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, 1998.
25. Launchbury, J. *Projections factorizations in partial evaluation*. Cambridge University Press, 1991.
26. Man Machine Systems. Design by contract for Java using JMSAssert. `http://www.mmsindia.com/`, 2000.
27. Matthews, J. and R. B. Findler. An operational semantics for R5RS Scheme. In *Workshop on Scheme and Functional Programming*, 2005.
28. McCarthy, J. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, 1961. `http://www-formal.stanford.edu/jmc/basis/basis.html`.
29. McFarlane, K. Design by contract framework. `http://www.codeproject.com/csharp/designbycontract.asp`.
30. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.
31. Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
32. Plösch, R. Design by contract for Python. In *IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference*, 1997. `http://citeseer.nj.nec.com/257710.html`.
33. Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.
34. Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. `http://homepages.inf.ed.ac.uk/gdp/publications/cbn_cbv_lambda.pdf`.
35. PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR05-4-v300, PLT Scheme Inc., 2005. `http://www.plt-scheme.org/techreports/`.
36. Reynolds, J. C. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
37. Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.
38. Sabry, A. and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.
39. Scott, D. S. Data types as lattices. *Society of Industrial and Applied Mathematics (SIAM) Journal of Computing*, 5(3):522–586, 1976.
40. The GHC Team. Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.
41. Wadler, P. and R. J. M. Hughes. Projections for Strictness Analysis. In Kahn, G., editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 385–407, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.
42. Xi, H., C. Chen and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the ACM Conference Principles of Programming Languages*, 2003.
43. Yang, Z. Encoding types in ml-like languages. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 1998.