Contracts as Pairs of Projections

Robert Bruce Findler, University of Chicago
Matthias Blume, Toyota Technological Institute at Chicago
University of Chicago Technical Report 2005-15

Summary. Assertion-based contracts provide a powerful mechanism for stating invariants at module boundaries and for enforcing them uniformly. Recently Findler and Felleisen have shown how to add contracts to higher-order functional languages, allowing programmers to assert invariants about functions as values.

In this paper, we develop a model for such contracts. Specifically, we follow Dana Scott's program and interpret software 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. For example, it shows how a contract that puts no obligation on either party is not the same as the most permissive contract for 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 [27]. In the 1980s, Meyer developed an entire philosophy of software design based on contracts, embodied in his object-oriented programming language Eiffel [26]. Nowadays, contracts are available in one form or another for many programming languages (*e.g.*, C [32], C++ [29], C# [25], Java [1, 4, 6, 18, 20–22], Scheme [12, 15], Smalltalk [3], Perl [5], and Python [28]). 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 the C and C++ entries to the 2005 ICFP programming contest used assertions [8], 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 }
```

[1] `http://bugs.sun.com/bugdatabase/top25_rfes.do` as of 10/16/2005

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. Since 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`, since `f` called its argument with a valid input. Instead, the blame for the violation must lie with `f`'s caller, since 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 negative positions in the contract (those positions that occur to the left of an odd number of 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.

---

[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 [11, 14]. 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.

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

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 earlier contract checker is in fact a disguised version of projections. Section 4 introduces pro-

jections 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 remainder of this paper, we focus on an idealized, pure version of Scheme [19, 23], 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 [7]. 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.

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 evaluation 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

An implementation 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 [12], we provided an implementation, where the contract construction functions were just record constructors and the interesting code was in the function that applied the contract, as shown in figure 2. The $\texttt{flat}_\texttt{l}$ and $\texttt{ho}_\texttt{l}$ functions collect their arguments. The $\texttt{guard}_\texttt{l}$ function is defined in cases based on the 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 $\texttt{guard}_\texttt{l}$ 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 this would not type-check

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

(define (flat_l p) p)
(define (ho_l dom rng) (cons dom rng))

(define (guard_l 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)
             (guard_l rng
                      (val (guard_l dom x neg pos))
                      pos
                      neg))
           (blame pos)))]))
```

Fig. 2. Original contract library implementation

in SML or Haskell. It does type-check, however, if we use the generalized abstract datatype [17, 36] `Contract_l`, shown as a comment in figure 2.

The `Contract_l` datatype constructors can be viewed as two defunctionalized functions [31], and `guard_l` as the defunctionalized version of apply. To re-functionalize the program, we can move the code in the first **`cond`** clause of `guard_l` 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 `guard_l` by a function application. The new type for contracts is thus a function that accepts all of the arguments that `guard_l` accepts (except the contract itself), and produces the same result that `guard_l` 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.

These two transformations of our contract implementation 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. We changed PLT Scheme's contract library from an implementation based on the first version of the code above to one based on the third version and, in the new library, checking a simple higher-order contract in a tight loop runs three times faster than it did before the change.

```
;; 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

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 [35] on a 1.25 GHz PowerPC G4, the projection 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 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 [34].

This correspondance 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 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 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. That is, the contract may replace some parts of its input with errors, but it must not change any other behavior.[4]

Projections have a natural ordering, as defined by Scott [34]

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

This ordering relates two projections 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 projection that ignores its argument and always signals an error is the smallest projection (*i.e.*, it likes the fewest values), and the identity function is the largest projection (*i.e.*, it likes the most values).

Given this ordering and the $\texttt{ho}_2$ contract combinator, it is natural to ask if the ordering is contra-variant in the domain and co-variant in the range of $\texttt{ho}_2$, analogous to conventional type systems. Surprisingly, as noted by Scott, it is co-variant in the domain.

Theorem 1. *(Scott [34]) For any projections,* $\texttt{d}_1$, $\texttt{d}_2$, *and* $\texttt{r}$, *if* $\texttt{d}_1 \ll \texttt{d}_2$, *then* $(\texttt{ho}_2\ \texttt{d}_1\ \texttt{r}) \ll (\texttt{ho}_2\ \texttt{d}_2\ \texttt{r})$.

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

```
    (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)))))         ;; composition
             ((λ (f) (λ (x) (r (f (d₂ x)))))        ;; and β_v
              f)))
=   (λ (f) ((λ (f) (λ (x) (r (f (d₁ x)))))         ;; β_v
             (λ (x) (r (f (d₂ x))))))
=   λ (f) (λ (x) (r ((λ (x) (r (f (d₂ x)))) (d₁ x)))) ;; β_v
=   (λ (f) (λ (x) (r (r (f (d₂ (d₁ x))))))))       ;; β_ω [33]
=   (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
=   (ho₂ d₁ r)                                     ;; definition of ho₂ □
```

[4] Technically, Scott's projections only add $\bot$, but errors are a better match for our work.

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

(define (flat₃ f)
  (cons (λ (x) (if (f x) (blame) x))   ;; see below
        (λ (x) x)))

(define (ho₃ a b)
  (let ([ae (car a)]
        [ac (cdr a)]
        [be (car b)]
        [bc (cdr b)])
    (cons (λ (f)
            (if (procedure? f)
                (λ (x) (be (f (ac x))))
                (blame)))  ;; see below
          (λ (f)
            (if (procedure? f)
                (λ (x) (bc (f (ae x))))
                f)))))
```

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

Thus, since 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 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 only assigns blame to the client and one that only assigns blame to the server, and then we can compare the projections separately. So, we represent contracts as pairs of projections, where 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. Figure 4 contains 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 projection (`ac`) in the server projection of `ho`'s result (the `car` position) and using the server's part of the domain projection (`ae`) in the client projection in the result (the `cdr` position). As shown earlier [10], for any projections `a`, `b`, `c`, and `d`,

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

where the ho_2 is from figure 3 and the ho_3 is from figure 4.

In an implementation, each projection would also be parameterized over some symbolic information to identify the guilty party more clearly, but here we use the positions of the projections in the pair to determine guilt. Parameterizing the projections is

straightforward, but makes the code more difficult to read, so we have omitted it here. Still, we can define a version of `guard` that ignores `pos` and `neg`:

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

and, if we had properly parameterized the projections in `ctc`, we would simply pass `pos` and `neg` to elements of the `ctc` pair.

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.

$$\texttt{(cons}\ a_s\ a_c\texttt{)} \lll \texttt{(cons}\ b_s\ b_c\texttt{)} \quad \textit{if and only if} \quad a_s \ll b_s \quad \textit{and} \quad b_c \ll a_c$$

Theorem 2. *The relation* $\lll$ *is a partial order.*

Proof. Follows directly from the definition and fact that $\ll$ is a partial order (Scott [34]).

Theorem 3. *For any projections,* $\texttt{d}_1$, $\texttt{d}_2$, *and* `r`, *if* $\texttt{d}_1 \lll \texttt{d}_2$, *then* $\texttt{(ho}_3\ \texttt{d}_2\ \texttt{r)} \lll \texttt{(ho}_3\ \texttt{d}_1\ \texttt{r)}$.

Proof (sketch). The proof is an algebraic manipulation in Sabry and Felleisen's equational theory $\lambda\beta_v X$ [30, 33] (without η_v) extended with δ rules for **`if`** [24] and C_{lift} [7] 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 the full details, see appendix A. □

In short, a blame-sensitive ordering on projections 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 $\lll$ ordering on projections 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]

```
c = c → c | (λ (x) e)
```

[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 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. `(λ (x) e)` *where, for each value* $\texttt{v}_1$ *in **Safe**, either* $\texttt{((λ (x) e) v}_1\texttt{)} \longrightarrow^* \texttt{v}_2$ *and* $\texttt{v}_2$ *is in **Safe**, or* $\texttt{((λ (x) e) v}_1\texttt{)}$ *diverges.*

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

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

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

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

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

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

Definition 4. $(\!|\cdot|\!) : \texttt{c} \rightarrow \texttt{e}$

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

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

Theorem 4. *For any c, c′:* $(\!|c|\!) \lll (\!|c'|\!) \;\Rightarrow\; [\![c]\!] \subseteq [\![c']\!]$

The proof is given in appendix B.

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

`(λ (x) false)` $\rightarrow$ `(λ (x) false)` `(λ (x) false)` $\rightarrow$ `(λ (x) true)`

In both cases, the range contract is irrelevant, since the domain contract always rejects

all values. Accordingly, they both map to the same set of values under $[\![\cdot]\!]$. The corresponding pairs of 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, $\texttt{p}_2 \not\ll \texttt{p}_1$ 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, since functions that always fail when applied are not generally useful. To express this restriction formally, we define a sub-language of c, called $\hat{\texttt{c}}$:

$$\begin{aligned} \hat{\texttt{c}} &= \texttt{ne-c} \mid \texttt{(λ (x) e)} \\ \texttt{ne-c} &= \texttt{ne-c} \to \hat{\texttt{c}} \mid \texttt{non-empty-predicate} \end{aligned}$$

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

Theorem 5.

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

Proof (sketch). The first part follows from the example above. The proof of the second part is given in appendix B.

6 Revisiting the Blume-McAllester example

Now that we have developed an ordering on contracts and can treat contracts as 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`*, since* `any` *should be the highest contract in the subtyping ordering. Accordingly, there is no way that* `(λ (x) x)` *should be blamed.*

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

```
(((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

> *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), since its contract is* `any`*. So, the expression* `(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 interpreation 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 $\ll$ ordering, `(cons (λ (x) x) (λ (x) (`**`blame`**`)))`. With this interpretation of `any`, the client is immediately blamed, as they predict.

To support Findler & Felleisen's answer, we must interpret `any` as the contract that never assigns blame, `(cons (λ (x) x) (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, since it will always immediately abort the computation with a contract violation. The contract that never assigns blame, however, is useful since 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, since 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.

Beyond giving us the mathematical tools to resolve this subtlety, the contracts-as-projections viewpoint has enabled us to build better models for contracts [10], to use contracts to connect nominal and structural type systems in a single language [13], and to interoperate between Java and Scheme [16]. We believe that this work is just the tip of the iceberg and intend to explore this point of view further.

Acknowledgments. Thanks to Bob Harper for alerting us to the connection between contracts and projections and to Matthias Felleisen for his 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/lib/Class/Contract/Production.pm`.
6. Duncan, A. and U. Hölzle. Adding contracts to Java with handshake. Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998.
7. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, pages 235–271, 1992.
8. Findler, R. B., E. Barzilay, M. Blume, M. Codik, M. Felleisen, M. Flatt, H.-H. Huang, J. Matthews, J. McCarthy, S. Owens, D. Press, M. Rainey, J. Reppy, J. Riehl, J. Spiro, D. Tucker and A. Wick. The eighth annual ICFP programming contest. `http://icfpc.plt-scheme.org/`.
9. Findler, R. B. and M. Blume. Contracts as pairs of projections. Technical Report TR-2005-15, University of Chicago Computer Science Department, 2005. `http://www.cs.uchicago.edu/research/publications/techreports/TR-2005-15`.

10. 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.
11. Findler, R. B. and M. Felleisen. Contract soundness for object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.
12. Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2002.
13. 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.
14. Findler, R. B., M. Latendresse and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ACM Conference Foundations of Software Engineering*, 2001.
15. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. http://www.mzscheme.org/.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. Kramer, R. iContract: The Java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, 1998.
22. Man Machine Systems. Design by contract for Java using JMSAssert. `http://www.mmsindia.com/`, 2000.
23. Matthews, J. and R. B. Findler. An operational semantics for R5RS Scheme. In *Workshop on Scheme and Functional Programming*, 2005.
24. 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`.
25. McFarlane, K. Design by contract framework. `http://www.codeproject.com/csharp/designbycontract.asp`.
26. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.
27. Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
28. 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`.
29. Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.
30. 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`.
31. 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.
32. Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.
33. Sabry, A. and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.
34. Scott, D. S. Data types as lattices. *Society of Industrial and Applied Mathematics (SIAM) Journal of Computing*, 5(3):522–586, 1976.
35. The GHC Team. Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.
36. Xi, H., C. Chen and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the ACM Conference Principles of Programming Languages*, 2003.

A Proof of theorem 3

Proof. Assume that $d_1 \lll d_2$ where

```
(define d1 (cons dc1 de1))
(define d2 (cons dc2 de2))
```

Thus, we know that

$de_1 \ll de_2$
$dc_2 \ll dc_1$

We want to show that, for an arbitrary contract `r`, where

```
(define r (cons rc re))
```

that

```
(ho d2 r) ⋘ (ho d1 r)
```

or, equivalently (from the definition of $\lll$),

```
(car (ho d2 r)) ≪ (car (ho d1 r))
(cdr (ho d1 r)) ≪ (cdr (ho d2 r))
```

Expanding the definition of `ho`, we get:

```
(ho d2 r) = (cons
              (λ (f)
                (if (procedure? f)
                    (λ (x) (re (f (dc2 x))))
                    (blame)))
              (λ (f)
                (if (procedure? f)
                    (λ (x) (rc (f (de2 x))))
                    f)))
(ho d1 r) = (cons
              (λ (f)
                (if (procedure? f)
                    (λ (x) (re (f (dc1 x))))
                    (blame)))
              (λ (f)
                (if (procedure? f)
                    (λ (x) (rc (f (de1 x))))
                    f)))
```

The lemmas 1 and 2 work out the two inequations, using these equations:

```
(v (if e1 e2 e3)) = (if e1 (v e2) (v e3))
;; McCarthy [24]

(if e1 e2 (if e1 e3 e4)) = (if e1 e2 e4)
```

```
;; McCarthy [24]

(v (blame)) = (blame)
;; Felleisen and Hieb [7] C_lift

((λ (x) E[(v x)]) e) = E[(v e)]  ;; x not free in E
;; Sabry and Felleisen [33] β_ω

((λ (x) e) v) = e[x/v]
;; Plotkin [30] β_v
```

and the retract rule, *i.e.*, for any retract value p,

```
(p (p e)) = (p e)
```

□

Lemma 1.

```
(cdr (ho d1 r)) ∝ (cdr (ho d2 r))
```

Proof.

```
   (cdr (ho3 d1 r)) ∘ (cdr (ho3 d2 r))
=  (λ (f)                                          ;; definition of ho3
     (if (procedure? f)
         (λ (x) (rc (f (de1 x))))
         f))
   ∘
   (λ (f)
     (if (procedure? f)
         (λ (x) (rc (f (de2 x))))
         f))
=  (λ (f)                                          ;; definition of
     ((λ (f)                                       ;; composition
        (if (procedure? f)
            (λ (x) (rc (f (de1 x))))
            f))
      ((λ (f)
         (if (procedure? f)
             (λ (x) (rc (f (de2 x))))
             f))
       f)))
=  (λ (f)                                          ;; β_v
     ((λ (f)
        (if (procedure? f)
            (λ (x) (rc (f (de1 x))))
            f))
      (if (procedure? f)
          (λ (x) (rc (f (de2 x))))
          f)))
```

```
=  (λ (f)                                          ;; lifting if
     (if (procedure? f)
         ((λ (f)
            (if (procedure? f)
                (λ (x) (rc (f (de1 x))))
                f))
          (λ (x) (rc (f (de2 x)))))
         ((λ (f)
            (if (procedure? f)
                (λ (x) (rc (f (de1 x))))
                f))
          f)))
=  (λ (f)                                          ;; βv twice
     (if (procedure? f)
         (if (procedure?
              (λ (x) (rc (f (de2 x)))))
             (λ (x)
               (rc ((λ (x) (rc (f (de2 x))))
                    (de1 x))))
             (λ (x)
               (rc (f (de2 x)))))
         (if (procedure? f)
             (λ (x) (rc (f (de1 x))))
             f)))
=  (λ (f)                                          ;; procedure?
     (if (procedure? f)                            ;; primitive
         (if #t
             (λ (x)
               (rc ((λ (x) (rc (f (de2 x))))
                    (de1 x))))
             (λ (x)
               (rc (f (de2 x)))))
         (if (procedure? f)
             (λ (x) (rc (f (de1 x))))
             f)))
=  (λ (f)                                          ;; if test true
     (if (procedure? f)
         (λ (x)
           (rc ((λ (x) (rc (f (de2 x))))
                (de1 x))))
         (if (procedure? f)
             (λ (x) (rc (f (de1 x))))
             f)))
```

```
= (λ (f)                                          ;; eliminate
    (if (procedure? f)                            ;; redundant
        (λ (x)                                    ;; procedure test
          (rc ((λ (x) (rc (f (de2 x))))
               (de1 x))))
        f))
= (λ (f)                                          ;; βω
    (if (procedure? f)
        (λ (x)
          (rc (rc (f (de2 (de1 x))))))
        f))
= (λ (f)                                          ;; retract law
    (if (procedure? f)
        (λ (x)
          (rc (f (de2 (de1 x)))))
        f))
= (λ (f)                                          ;; by assumption
    (if (procedure? f)                            ;; and lemma 3
        (λ (x) (rc (f (de1 x))))
        f))
= cdr (ho3 d1 r)                                  ;; definition of ho3
```

□

Lemma 2.

```
(car (ho d2 r)) ∝ (car (ho d1 r))
```

Proof.

```
    (car (ho3 d2 r)) ∘ (car (ho3 d1 r))
  = (λ (f)                                        ;; definition of ho3
      (if (procedure? f)
          (λ (x) (re (f (dc2 x))))
          (blame)))
    ∘
    (λ (f)
      (if (procedure? f)
          (λ (x) (re (f (dc1 x))))
          (blame)))
```

```
=  (λ (f)                                          ;; definition
     ((λ (f)                                       ;; of composition
        (if (procedure? f)
            (λ (x) (re (f (dc2 x))))
            (blame)))
      ((λ (f)
         (if (procedure? f)
             (λ (x) (re (f (dc1 x))))
             (blame)))
       f)))
=  (λ (f)                                          ;; βv
     ((λ (f)
        (if (procedure? f)
            (λ (x) (re (f (dc2 x))))
            (blame)))
      (if (procedure? f)
          (λ (x) (re (f (dc1 x))))
          (blame))))
=  (λ (f)                                          ;; lifting if
     (if (procedure? f)
         ((λ (f)
            (if (procedure? f)
                (λ (x) (re (f (dc2 x))))
                (blame)))
          (λ (x) (re (f (dc1 x)))))
         ((λ (f)
            (if (procedure? f)
                (λ (x) (re (f (dc2 x))))
                (blame)))
          (blame))))
=  (λ (f)                                          ;; abort rule
     (if (procedure? f)
         ((λ (f)
            (if (procedure? f)
                (λ (x) (re (f (dc2 x))))
                (blame)))
          (λ (x) (re (f (dc1 x)))))
         (blame)))
```

```
= (λ (f)                                          ;; β_v
    (if (procedure? f)
        (if (procedure?
             (λ (x) (re (f (dc1 x)))))
            (λ (x)
              (re ((λ (x)
                     (re (f (dc1 x))))
                   (dc2 x))))
            (blame))
        (blame)))
= (λ (f)                                          ;; procedure?
    (if (procedure? f)                            ;; predicate
        (if #t
            (λ (x)
              (re ((λ (x)
                     (re (f (dc1 x))))
                   (dc2 x))))
            (blame))
        (blame)))
= (λ (f)                                          ;; if test true
    (if (procedure? f)
        (λ (x)
          (re ((λ (x) (re (f (dc1 x))))
               (dc2 x))))
        (blame)))
= (λ (f)                                          ;; β_ω
    (if (procedure? f)
        (λ (x)
          (re (re (f (dc1 (dc2 x))))))
        (blame)))
= (λ (f)                                          ;; retract law
    (if (procedure? f)
        (λ (x)
          (re (f (dc1 (dc2 x)))))
        (blame)))
= (λ (f)                                          ;; assumption
    (if (procedure? f)                            ;; and lemma 3
        (λ (x)
          (re (f (dc2 x))))
        (blame)))
= (car (ho3 d2 r))                                ;; definition of ho3
```

□

Lemma 3. *For all retracts,* a *and* b*, if* a = a o b *then* a = b o a.

Proof.

	`a` = `a ∘ b`	
`a ∘`	`a` = `a ∘ b ∘ a`	compose `a` on right of both sides
	`a` = `a ∘ b ∘ a`	retract law on left
	`a` = `b ∘ a`	retract law on right □

B Proof of theorem 4

We want to show that for any c, c': $(\langle\!|c|\!\rangle \lll \langle\!|c'|\!\rangle) \Rightarrow (\llbracket c \rrbracket \subseteq \llbracket c' \rrbracket)$.

Proof: The proof proceeds by simultaneous induction on the structure of c and c'. There are four cases to consider:

$\boxed{c = (\lambda\ (\mathtt{x})\ \mathtt{e}), c' = (\lambda\ (\mathtt{x})\ \mathtt{e'})}$ Here $c \lll c'$ implies that `(λ (x) e)` must accept a value v if and only if `(λ (x) (and e e'))` accepts v.[6] Therefore, whenever `(λ (x) e)` accepts a given value, so does `(λ (x) e')`. The conclusion $\llbracket c \rrbracket \subseteq \llbracket c' \rrbracket$ follows directly from that.

$\boxed{c = c_1 \to c_2, c' = c'_1 \to c'_2}$ This is the most interesting case. Let us write c^+ for `(car` $\langle\!|c|\!\rangle$`)` and c^- for `(cdr` $\langle\!|c|\!\rangle$`)`. With this, the condition $\langle\!|c|\!\rangle \lll \langle\!|c'|\!\rangle$ can be reduced to two equations which have to hold for arbitrary function values f:

$$c_2{}^+ \circ f \circ c_1{}^- = c_2{}^+ \circ c'_2{}^+ \circ f \circ c'_1{}^- \circ c_1{}^-$$
$$c'_2{}^- \circ f \circ c'_1{}^+ = c'_2{}^- \circ c_2{}^- \circ f \circ c_1{}^+ \circ c'_1{}^+$$

Since f would be able to observe any difference between $c_1{}^-$ and $c'_1{}^- \circ c_1{}^-$ (which is the same as $c_1{}^- \circ c'_1{}^-$), it must be the case that $c_1{}^- \lll c'_1{}^-$, and likewise that $c'_1{}^+ \lll c_1{}^+$. Thus, we have $\langle\!|c'_1|\!\rangle \lll \langle\!|c_1|\!\rangle$, and by induction hypothesis $\llbracket c'_1 \rrbracket \subseteq \llbracket c_1 \rrbracket$.
For the second part we need to consider two cases: 1. If c'_1 is the empty contract, then the conclusion holds trivially since $c'_1 \to c'_2$ is then satisfied by every function value. 2. If c'_1 is non-empty, then there must be at least one value that is not mapped to error by $c'_1{}^+$. Moreover, the same is always true for $c_1{}^-$. (This fact does not depend on whether or not c_1 is empty but follows from the construction of `(cdr` $\langle\!|c_1|\!\rangle$`)`.)
Thus, we can consider the family of functions $f \in \{\,$`(λ (x) v)` $\mid$ `v` is a value$\}$. These functions are constant functions that ignore their arguments and produce a fixed value; there is one such function for each value. Since the equations are true for all functions including this family, we also find that $\langle\!|c_2|\!\rangle \lll \langle\!|c'_2|\!\rangle$ and therefore $\llbracket c_2 \rrbracket \subseteq \llbracket c'_2 \rrbracket$. The conclusion $\llbracket c_1 \to c_2 \rrbracket \subseteq \llbracket c'_1 \to c'_2 \rrbracket$ now follows directly from the definition of $\llbracket \cdot \rrbracket$.

$\boxed{c = c_1 \to c_2, c' = (\lambda\ (\mathtt{x})\ \mathtt{e'})}$ By definition of $c \lll c'$ we have

$$(\lambda\ (\mathtt{x})\ \mathtt{e'})^- = (\lambda\ (\mathtt{x})\ \mathtt{e'})^- \circ (c_1 \to c_2)^-$$

[6] We write `(and` $\mathtt{e}_1$ $\mathtt{e}_2$`)` for `(if` $\mathtt{e}_1$ $\mathtt{e}_2$ `#f)`.

Since $(\lambda\ (\mathtt{x})\ \mathtt{e}')^-$ is the identity function, this equation implies that for all functions f we have

$$c_2{}^- \circ f \circ c_1{}^+ = f$$

and, therefore, that $c_1{}^+$ and $c_2{}^-$ are identities. This implies that $\underline{\text{safe}} \ll c_1$ and $c_2 \ll \underline{\text{safe}}$.[7] Using the induction hypothesis we get $[\![\underline{\text{safe}}]\!] \subseteq [\![c_1]\!]$ and $[\![c_2]\!] \subseteq [\![\underline{\text{safe}}]\!]$, and therefore that $[\![c]\!]$ contains only safe functions:

$$[\![c]\!] = [\![c_1 \to c_2]\!] \subseteq [\![\underline{\text{safe}} \to \underline{\text{safe}}]\!].$$

From the definition of $c \ll c'$ we also get

$$(c_1 \to c_2)^+ = (c_1 \to c_2)^+ \circ (\lambda\ (\mathtt{x})\ \mathtt{e}')^+.$$

Abusing notation this can be rewritten to
$(\lambda\ (\mathtt{f})\ (\mathbf{if}\ (\mathtt{procedure?\ f})\ c_2{}^+ \circ \mathtt{f} \circ c_1{}^-\ (\mathbf{blame})))\ =$
$(\lambda\ (\mathtt{f})\ (\mathbf{if}\ (\mathbf{and}\ (\mathtt{procedure?\ f})\ ((\lambda\ (\mathtt{x})\ \mathtt{e}')\ \mathtt{f}))\ c_2{}^+ \circ \mathtt{f} \circ c_1{}^-\ (\mathbf{blame})))$
For this to hold it must be the case that $(\lambda\ (\mathtt{x})\ \mathtt{e}')$ accepts every function value. As a result, $[\![c']\!]$ contains *all* safe function values, and since $[\![c]\!]$ contains *only* safe function values we conclude that $[\![c]\!] \subseteq [\![c']\!]$.

$\boxed{c = (\lambda\ (\mathtt{x})\ \mathtt{e}), c' = c_1' \to c_2'}$ By definition of $c \ll c'$ we have

$$(\lambda\ (\mathtt{x})\ \mathtt{e})^+ = (\lambda\ (\mathtt{x})\ \mathtt{e})^+ \circ (c_1' \to c_2')^+.$$

Writing out the left-hand side yields

$$(\lambda\ (\mathtt{f})\ (\mathbf{if}\ ((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{f})\ \mathtt{f}\ (\mathbf{blame}))),$$

while the right-hand side is equivalent to

```
(λ (f) (if (procedure? f)
           (let ((g c2'+∘f∘c1'- ))
              (if ((λ (x) e) g) g (blame)))
           (blame)))
```

Now recall that the predicate in a flat contract, when presented with an argument that is a function, must not invoke its argument. This follows from the restriction that predicates must be total. Since there is no way of distinguishing between different functions by any means other than invoking them, predicates must treat all functions uniformly: if they return true for *any* function argument, then they must return true for *all* function arguments [10]. As a consequence, $((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{g})$ and $((\lambda\ (\mathtt{x})\ \mathtt{e})\ \mathtt{f})$ must be the same, so we can further rewrite the right-hand side to

```
(λ (f) (if (and (procedure? f) ((λ (x) e) f))
           c2'+ ∘ f ∘ c1'-
           (blame)))
```

[7] From here on we will write $\underline{\text{safe}}$ as a shorthand for the contract $(\lambda\ (\mathtt{x})\ \#\mathtt{t})$, i.e., the contract that is satisfied by precisely the values in **Safe**.

Since this has to be equal to `(λ (f) (if ((λ (x) e) f) f (blame)))` it must be the case that `(λ (x) e)` does not accept any non-function value. If `(λ (x) e)` never returns true, then the conclusion holds trivially.
If it is true for any function value (which means it is true for all function values), then $[\![\texttt{(λ (x) e)}]\!]$ is precisely the set of safe function values. In this case we also have for any function f that $f = {c'_2}^+ \circ f \circ {c'_1}^-$, i.e., both ${c'_2}^+$ and ${c'_1}^-$ are identities. Again, it is easy to see that this implies $c'_1 \ltimes\!\ll \underline{\text{safe}}$ and $\underline{\text{safe}} \ltimes\!\ll c'_2$. Using the induction hypothesis we get $[\![c'_1]\!] \subseteq [\![\underline{\text{safe}}]\!]$ and $[\![\underline{\text{safe}}]\!] \subseteq [\![c'_2]\!]$, and therefore

$$[\![\underline{\text{safe}} \to \underline{\text{safe}}]\!] \subseteq [\![c'_1 \to c'_2]\!].$$

Thus, in this case $[\![c'_1 \to c'_2]\!]$ is a superset of the safe function values, i.e., a superset of $[\![c]\!]$.

□

C Proof of theorem 5

We want to show that for arbitrary $\hat{c}$, $\hat{c}'$: $[\![\hat{c}]\!] \subseteq [\![\hat{c}']\!] \;\Rightarrow\; (\!|\hat{c}|\!) \ltimes\!\ll (\!|\hat{c}'|\!)$

As in the case of theorem 4, we prove this by simultaneous induction on the structure of $\hat{c}$ and $\hat{c}'$. Since the grammar for contracts ($\hat{c}$) is merely a refinement of the original grammar, it is sufficient to consider the same four cases as in the previous proof. (The second case will take advantage of the only difference between the two grammars, namely that in $\hat{c}$ no empty contract can ever appear as the domain of a function contract.)

$\boxed{\hat{c} = \texttt{(λ (x) e)}, \hat{c}' = \texttt{(λ (x) e')}}$ By definition, this means that for all $v \in$ **Safe**, whenever `(λ (x) e)` accepts v, then so does `(λ (x) e')`. Let $f \notin$ **Safe**. Such an f must be a function value. Pick some other function $g \in$ **Safe** and recall that neither of the two predicates is able to distinguish between f and g. Therefore, for all values v (safe or unsafe) we have that whenever `(λ (x) e)` accepts v then `(λ (x) e')` accepts v as well. From this the statement of the theorem follows immediately.

$\boxed{\hat{c} = \hat{c}_1 \to \hat{c}_2, \hat{c}' = \hat{c}'_1 \to \hat{c}'_2}$ First we show that $[\![\hat{c}_2]\!] \subseteq [\![\hat{c}'_2]\!]$ by considering values `(λ (x) v)` for $\texttt{v} \in [\![\hat{c}_2]\!]$. Since $\hat{c}_1$ is non-empty, all these values are in $[\![\hat{c}]\!]$ and therefore also in $[\![\hat{c}']\!]$. Since $\hat{c}'_1$ is also non-empty, this can only be the case if $[\![\hat{c}'_2]\!]$ contains every such v, so $[\![\hat{c}_2]\!] \subseteq [\![\hat{c}'_2]\!]$, and by induction hypothesis $(\!|\hat{c}_2|\!) \ltimes\!\ll (\!|\hat{c}'_2|\!)$.
The next step is to show that $[\![\hat{c}'_1]\!] \subseteq [\![\hat{c}_1]\!]$. Indirect: Suppose $v \in [\![\hat{c}'_1]\!] \setminus [\![\hat{c}_1]\!]$. Since contract checking is complete and contract guards are expressible in the language [2], there exists some $f =$ `(λ (x) b)` which, when applied to this v, calls **blame** or gets stuck, but which succeeds (e.g., by looping indefinitely) when applied to any $w \in [\![\hat{c}_1]\!]$. This means that $f \in [\![\hat{c}]\!]$ but also $f \notin [\![\hat{c}']\!]$, which is a contradiction. Therefore, $[\![\hat{c}'_1]\!] \subseteq [\![\hat{c}_1]\!]$ and by induction hypothesis $(\!|\hat{c}'_1|\!) \ltimes\!\ll (\!|\hat{c}_1|\!)$.
Since $\to$ is co-variant in the range and contra-variant in the domain we get the desired result $(\!|\hat{c}|\!) = (\!|\hat{c}_1 \to \hat{c}_2|\!) \ltimes\!\ll (\!|\hat{c}'_1 \to \hat{c}'_2|\!) = (\!|\hat{c}'|\!)$.

$\boxed{\hat{c} = \hat{c}_1 \to \hat{c}_2, \hat{c}' = \texttt{(λ (x) e')}}$ The set $[\![\hat{c}_1 \to \hat{c}_2]\!]$ contains at least one value (namely the function that ignores its argument and then loops forever). Therefore, `(λ (x)`

`e'`), which as before cannot distinguish between different functions, must accept not only this one function value but *every* function value. Therefore, by definition, the function values in $[\![\hat{c}']\!]$ are precisely the safe function values. Since $[\![\hat{c}]\!]$ contains *only* function values, all these function values must therefore be safe function values:

$$[\![\hat{c}_1 \to \hat{c}_2]\!] \subseteq [\![\underline{\text{safe}} \to \underline{\text{safe}}]\!]$$

This is an instance of the previous case, giving us:

$$(\!|\hat{c}_1 \to \hat{c}_2|\!) \lll (\!|\underline{\text{safe}} \to \underline{\text{safe}}|\!).$$

If we can show that $(\!|\underline{\text{safe}} \to \underline{\text{safe}}|\!) \lll (\!|\texttt{(}\lambda\ \texttt{(x) e')}|\!)$, then the desired result follows from transitivity of $\lll$.
To show that $(\!|\underline{\text{safe}} \to \underline{\text{safe}}|\!) \lll (\!|\texttt{(}\lambda\ \texttt{(x) e')}|\!)$ we simply check the definition. Two equations must hold for this to be true:

$$(\underline{\text{safe}} \to \underline{\text{safe}})^+ = (\underline{\text{safe}} \to \underline{\text{safe}})^+ \circ \texttt{(}\lambda\ \texttt{(x) e')}^+$$
$$\texttt{(}\lambda\ \texttt{(x) e')}^- = \texttt{(}\lambda\ \texttt{(x) e')}^- \circ (\underline{\text{safe}} \to \underline{\text{safe}})^-$$

The first equation holds because, as we have argued above, `(`λ `(x) e')` accepts all function values. For the second equation it suffices to check that $(\underline{\text{safe}} \to \underline{\text{safe}})^-$ is the identity projection.

$\boxed{\hat{c} = \texttt{(}\lambda\ \texttt{(x) e)}, \hat{c}' = \hat{c}'_1 \to \hat{c}'_2}$ All values in $[\![\texttt{(}\lambda\ \texttt{(x) e)}]\!]$ must be functions, so `(`λ `(x) e)` does not accept any non-function value. Since it is unable to distinguish between different function values, there are only two possible cases for `(`λ `(x) e)`: either it is equivalent to `(`λ `(x) #f)` or it behaves like `(`λ `(x) (procedure? x))`.
If $\hat{c}$ is `(`λ `(x) #f)` we show that $(\!|\texttt{(}\lambda\ \texttt{(x) \#f)}|\!) \lll (\!|\hat{c}'_1 \to \hat{c}'_2|\!)$ simply by plugging it into the definition, observing that the following two equations hold:

$$\texttt{(}\lambda\ \texttt{(x) \#f)}^+ = \texttt{(}\lambda\ \texttt{(x) \#f)}^+ \circ (\hat{c}'_1 \to \hat{c}'_2)^+$$
$$(\hat{c}'_1 \to \hat{c}'_2)^- = (\hat{c}'_1 \to \hat{c}'_2)^- \circ \texttt{(}\lambda\ \texttt{(x) \#f)}^-$$

If $\hat{c}$ is `(`λ `(x) (procedure? x))`, then $[\![\hat{c}]\!]$ is precisely the set of safe functions, i.e., $[\![\underline{\text{safe}} \to \underline{\text{safe}}]\!] \subseteq [\![\hat{c}'_1 \to \hat{c}'_2]\!]$. Again, this is an instance of the second case, so $(\!|\underline{\text{safe}} \to \underline{\text{safe}}|\!) \lll (\!|\hat{c}'_1 \to \hat{c}'_2|\!)$. Using the definition of $\lll$ it is straightforward to see that $(\!|\texttt{(}\lambda\ \texttt{(x) (procedure? x))}|\!) \lll (\!|\underline{\text{safe}} \to \underline{\text{safe}}|\!)$. By transitivity of $\lll$ we obtain the desired result:

$$(\!|\hat{c}|\!) = (\!|\texttt{(}\lambda\ \texttt{(x) (procedure? x))}|\!) \lll (\!|\underline{\text{safe}} \to \underline{\text{safe}}|\!) \lll (\!|\hat{c}'_1 \to \hat{c}'_2|\!) = (\!|\hat{c}'|\!)$$

□