An Investigation of Contracts as Projections

Robert Bruce Findler
University of Chicago

Matthias Blume
Toyota Technical Institute

Matthias Felleisen
Northeastern University

Abstract (April 1, 2004 version)

Software contracts help programmers enforce program properties that the language's type system cannot express. Unlike types, contracts are (usually) enforced at run-time. When a contract fails, the contract system signals an error. Beyond such errors, contracts should have no other observable (functional) effect on the program's results. In most implementations, however, the language of contracts is the full-fledged programming language, which means that programmers may (intentionally or unintentionally) introduce visible effects into their contracts.

Here we present the results of investigating the nature of contracts from a denotational perspective. Specifically, we use SPCF and the category of observably sequential functions to show that contracts are best understood as projections. Thus far, the investigation has produced a significantly faster contract implementation and the insight that our contract language cannot express all projections, which in turn has produced a new contract combinator.

1 Modeling Contracts

Many programming languages support dynamically enforced software contracts [1, 2, 6, 8, 9, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26, 28]. With software contracts, programmers can state program invariants that the type system cannot express. Unlike types, however, contracts are not verified at compile time but monitored at run-time.[1] When the contract monitor discovers that a contract does not hold for a particular function argument or result, it signals an error. Thus, contracts impose an overhead on the execution of programs, and contract system designers therefore provide a mechanism to disable contract monitoring.

The nature of contract systems suggests that contracts should not affect the functional behavior of programs. Otherwise, running a program without contracts may produce different results than running programs with contracts, even without any contract violations. Unfortunately, ensuring that contracts have no observable effects is a difficult task. Therefore the designers of software contracts simply ignore the problem and have programmers formulate contracts in the full-fledged programming language.

To improve our understanding of this problem, we developed an operational model of a functional programming language with contracts, λ^{CON} [9]. That model specifies the meaning of the language and the meaning of contracts. In particular, it shows how to blame a specific program component for a contract violation.

In this paper we present the results of investigating this setting from a denotational perspective. Specifically, we develop a denotational model of contracts using SPCF and the category of observably sequential functions [3, 4]. We chose SPCF because it has a purely functional ("standard") model that is fully abstract and yet includes errors, exceptions, and a modicum of exception handling. It is thus possible to study contracts in a minimally realistic setting with pleasant mathematical properties.

The goal of this effort is to characterize contracts mathematically and to use this characterization to improve contract systems. We start from the conjecture that software contracts are *projections* and then ask whether all syntactic contracts denote projections and whether our contract syntax expresses all projections. The answers—though not surprising in retrospect—have already improved the implementation of our contract system [10].

The paper's organization follows this introduction. The next section informally explains contracts and why we use projections to model them. Then we recall the Cartesian-closed category of manifestly sequential functions and introduce SPCF, its syntax, types, and semantics. Based on this prelude, we formally introduce projections as contracts, answer the above questions, and discuss the implications for our implementation of contracts.

2 Contracts as Projections

A contract in an imperative programming language consists of a pair of assertions for a procedure, *i.e.*, a precondition on the arguments of the procedure and the state

[1] Extended static checking [7] and soft typing [5] are efforts to validate contracts statically and to generate code to check those parts of the contract that cannot be verified.

of the world at call-time and a post condition on the result of the function and the state at the time of the return. For example, here is a contract for a function that computes the square root of a real number (with C-like syntax):

```
// @pre:  (i >= 0.0)
// @post: (@result - i * i <= 0.001)
double sqrt(double i) { ... }
```

A naive contract system may translate this kind of contract into a pair of assertions in the procedure's body:

```
double sqrt(double i) {
 double r;
 assert("blame sqrt's caller", i >= 0.0);
 r = true_sqrt_function(i);
 assert("blame sqrt", r - i * i <= 0.001);
 return r;
}
```

Pre- and post-conditions translate into "filter" functions for a functional language (with Scheme-like syntax):

```
(define (sqrt i) (double →double)
  (range "blame sqrt" (λ(r) (≤ (abs (− i (∗ r r))) .001))
    (true-sqrt-function
      (positive "blame sqrt's caller" i))))
```

A filter function checks that its argument is in some subset and, if so, returns it. Otherwise it raises an error. The *positive* filter, for example, is defined as

```
(define (positive msg candidate) (string double →double)
  (if (≥ candidate 0.0) candidate (error msg)))
```

In object-oriented languages, a programmer adds contracts to (the equivalent of Java) interfaces:

```
interface IQueue {
  boolean empty();
  // @post: (!this.empty())
  void enq(double i);
  // @pre: (!this.empty())
  double deq();
}
```

Class definitions can specify that they implement `IQueue`:

```
class Queue implements IQueue { ... }
class FastQueue implements IQueue { ... }
class RemoteQueue implements IQueue { ... }
```

This simply means that the assertions for `IQueue` are imposed on the methods in each class.

From a foundational perspective, the examples suggest three things: (1) contracts are functions, (2) contracts are idempotent on good values, and (3) they may raise errors. Readers with some basic acquaintance in denotational semantics recognize these conditions as (roughly) Scott's [29] "types as retracts" idea. Indeed, contracts are not just retracts but projections—a special retract, which means that a contract c is a function that satisfies these two equations:

$$c \circ c = c \qquad\qquad c \sqsubseteq 1^e .$$

The e superscript on the identity function suggests that it may return an error instead of a proper value. In short, we conjecture that *contracts are (error) projections.*

Informally, this conjecture holds when we move from flat values, *e.g.*, *double*s, to functions. In a functional language with contracts, we may define *sqrt* as follows:

(define (*sqrt i*) (*double*[*positive*] →*double*[*positive*]) $\cdots$)

As already seen, the contracts on the domain and range of this function become filters (or coercions) on *double*s. In general, a contract for functions has the shape

(*domain-contract* → *range-contract*)

and, when applied to a function *f*, produces

(λ(*x*) (*range-contract* (*f* (*domain-contract x*))))

Similarly, when we deal with objects and callbacks on objects[2] the contract system distributes filters into the object's methods, and the distributed filters observe contract failures for ground types. In short, Scott's equations for types as retracts apply to our treatment of contracts and suggest that the conjecture is a fruitful starting point for our exploration.

3 SEQ: Manifestly Sequential Functions

Formulating the semantics of a programming language based on the simply-typed λ-calculus requires a Cartesian-closed category (CCC). For a language like PCF that includes primitives and recursion, we actually need a cpo-enriched CCC. Equipped with such a CCC, we can interpret type judgments of the language as arrows in the category.

In this section, we review the basic building blocks of $\mathcal{SEQ}$, the Cartesian-closed category of manifestly sequential functions, and $\mathcal{S}eq$ an isomorphic category where functions are uniquely represented as trees. We focus on the four essential elements in the construction of the category: the underlying mechanism for generating domain elements as trees; the construction of the exponent, *i.e.*, the function trees; the application of functions to arguments; and the generation of a manifestly sequential domain. Our goal is merely to recall the relevant aspects of observably sequential domains [3].

We represent trees as sets of paths. Each path describes how the function would interact with a particular context. They are generated from atomic pieces: addresses and data. Consistency requirements govern the paths so that each function maps to a unique tree, and each tree has a unique interpretation as a function.

[2]An object A that receives another object B and calls one of B's method employs (the widely used) callback mechanism. This is the OO equivalent of higher-order functions in functional languages.

Definition 1 (Sequential Data Structure) *A sequential data structure (*sds*) S is a 3-tuple (A, D, P) consisting of three countable sets: a set A of* addresses*; a set D of* data*; and a prefix-closed set P of non-empty paths that alternate in A and D, starting with an element of A.* **Notation:** *Paths are written as addresses and data separated by periods.*

A query *is a path that ends in an address and a* response *is a path that ends in a datum. The set of queries for an sds S is written Que_S and the set of responses for S is written Res_S.*

Each address in an sds corresponds to a unique aspect of the class of values that the sds represents. Each datum corresponds to the instantiation of an aspect for a particular value. For example, the sds for the natural numbers, $\mathbf{N}$, has one address: $?$, since there is only one interesting aspect of a number: its value. Accordingly, $\mathbb{N}$ is the set used for data in $\mathbf{N}$ and its paths are $\{?, ?.n | n \in \mathbb{N}\}$. The only query for $\mathbf{N}$ is $?$. Each path of the shape $?.n$ is a response.

Definition 2 (Manifestly Sequential Domain) *Let S be an sds and let $\mathbb{E} = \{\mathsf{e}_1, \mathsf{e}_2, ...\}$ be a countable set of errors, disjoint from the data in S, with one distinguished element* ce*, called a contract error.*

An observable response r is an element of

$$Res_S^{\mathbb{E}} \overset{def}{=} Res_S \cup \{q.\mathsf{e} | q \in Que_S, \mathsf{e} \in \mathbb{E}\}.$$

An element $t \in \mathbb{D}(S)$ (the domain associated with the sds S) is a set of observable responses that is closed under the prefix ordering and under greatest lower bounds:

- *if $r \in t$ and r' is a subpath of r, then that $r' \in t$, and*
- *if $r, r' \in t$ than the greatest lower bound path of r and r' is in $Res_S^{\mathbb{E}}$ (and, by prefix closure, is also in t).*

The set of elements of $\mathbb{D}(S)$ is ordered by set inclusion. The domain element $PC(p)$ is the one generated by prefix closure of the path p.

A function from $\mathbb{D}(S_1)$ to $\mathbb{D}(S_2)$ gradually explores the paths in its argument. The exploration proceeds as an exchange between the function and the argument. The function issues a query asking to know the next link on a path. The argument sends back a response, extending the query with one piece of data. This response may let the function determine its own output or, if the argument itself is a function, it may request information about its input. It may also be an error response; if so, the the error must be propogated.

Consider a function on $\mathbf{N}$. It simply asks for the datum at the single address. Once it knows that number, it knows everything about its input. For $\mathbf{N} \longrightarrow \mathbf{N}$, the story is different. The input is now a tree with many paths, which means that the function can explore more than a single path. Indeed, it generally explores a finite portion of the tree. Still, the exploration is sequential and therefore corresponds to a linear sequence of queries and responses.

Definition 3 (Path Sequence) *A* path sequence *s over an sds S is an alternating sequence of queries (Que_S) and responses (Res_S) such that:*

1. *the path does not duplicate elements in Que_S or Res_S;*
2. *$s_0 \in A$ (that is, s_0 has length 1); and*
3. *for all $i \geq 1$ such that $2i + 1 \leq |s|$, there exists $d \in D$ such that $s_{(2i+1)} = s_{(2i)} \cdot d$;*
4. *for all $i \geq 1$ such that $2i \leq |s|$ there exists an j such that $2j+1 < 2i$ and $s_{(2i)} = s_{(2j+1)} \cdot a$ for some $a \in A$.*

$\|s\|$ denotes the finite tree of responses in a path sequence.

The conditions guarantee that each path sequence starts with an initial query; that each response immediately follows its corresponding query; and that any query in the sequence has all of its shorter queries in the sequence first.

Definition 4 (Exponent sds) *Let $S_1 = (A_1, D_1, P_1)$ and $S_2 = (A_2, D_2, P_2)$ be* sds*'s. Let $Res_1 = Res_{S_1}$, $Que_1 = Que_{S_1}$, and $\mathscr{S}_1$ be the set of path sequences over S_1. The* exponent sds *$S_1 \Rightarrow S_2$ is (A, D, P) where*

$$\begin{aligned}
A &= Res_1 \uplus A_2 \\
D &= Que_1 \uplus D_2 \\
P &= \{p \in (A, D)^* | \\
&\qquad p_0 = \langle a, 2\rangle \\
&\qquad \pi_1^{\Rightarrow}(p) \in \mathscr{S}_1, \\
&\qquad \pi_2^{\Rightarrow}(p) \in P_2 \text{ or } \pi_2^{\Rightarrow}(p) = \epsilon, \\
&\qquad \text{if } p_{(i+1)} = \langle a, 2\rangle, \text{ then } p_i = \langle d, 2\rangle, \\
&\qquad \text{if } p_{(i+1)} = \langle r, 1\rangle, \text{ then } p_i = \langle q, 1\rangle, \\
&\qquad \text{where } a \in A_2, d \in D_2, r \in Res_1, q \in Que_1\}.
\end{aligned}$$

The functions $\pi_1^{\Rightarrow} : P \longrightarrow (Que_1 \cup Res_1)^*$ and $\pi_2^{\Rightarrow} : P \longrightarrow (A_2 \cup D_2)^*$ project a path to path sequences and paths, respectively. The extensions of these functions to paths in domain elements propagate errors. **Notation:** *We use $\langle \cdot, 1\rangle$ and $\langle \cdot, 2\rangle$ to inject objects into the disjoint unions of A and D in an exponent sds.*

Consider $\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.3, 1\rangle.\langle 4, 2\rangle$, a path in $\mathbf{N} \Rightarrow \mathbf{N}$. The initial element in the path is always an announcement that the function is about to produce some output. The second link asks for the function's input. The third one indicates that this path describes what happens when the input is 3; a function that contains this path returns 4 in this case. To generalize this path to a tree that represents the **add1** function, we write $\{\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n, 1\rangle.\langle n+1, 2\rangle | n \in \mathbb{N}\}$. Each path returns a number that is one larger than the input.

Functions that do not consider their input have shorter paths. For example, $\{\langle ?, 2\rangle.\langle 3, 2\rangle\}$ is the tree that represents the constant function that returns 3.

Functions on natural numbers do not exploit the full generality of the exponent construction. After all, there is only a single aspect of the context that it can explore, namely the value of its input. Functions that accept functions on natural numbers, however, do need the ability to explore multiple different paths in their inputs before producing any output. Consider this path in $(\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$:

$$\begin{array}{lcll}
\langle ?, 2\rangle & . & \langle\langle ?, 2\rangle, 1\rangle & (2)\\
 & . & \langle\langle ?, 2\rangle.\langle ?, 1\rangle, 1\rangle & (3)\\
 & . & \langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.3, 1\rangle, 1\rangle & (4)\\
 & . & \langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.3, 1\rangle.\langle 4, 2\rangle, 1\rangle & (5)\\
 & . & \langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.4, 1\rangle, 1\rangle & (6)\\
 & . & \langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.4, 1\rangle.\langle 6, 2\rangle, 1\rangle & (7)\\
 & . & \langle 10, 2\rangle & (8)
\end{array}$$

The first link is again an announcement that the function will produce a result. The second one asks the function's argument for its result. On this path, the argument function then responds by asking, in turn, for its input. This path uses 3 and then shows what happens when the argument function returns 4. Put differently, this part of the path (up to step (5)) shows how the above path in **add1** is used in a function that consumes functions from $\mathbf{N}$ to $\mathbf{N}$—it is just embedded in $\langle\cdot, 1\rangle$. The two steps (7) and (8) show the function asking its argument what it returns for the input 4 and then being told that the result of that call is 6. Then the function returns 10 in the final step. In short, any function that contains this path produces 10 when passed a function f such that $f(3)=4$ and $f(4)=6$.

Each domain element is a set of responses that correspond to the answers for some set of queries (the answered queries). In addition, there are a set of queries for which the domain element diverges (the open queries).

Definition 5 (Open, Answered Queries) *For an sds $S = (A, D, P)$, and $x \in \mathbb{D}(S)$, a query $q \in Que_S$ is:*

- *answered ($q \in Answered(x)$) if $q \sqsubseteq r$ for some $r \in x$.*
- *open ($q \in Open(x)$) if $q \notin Answered(x)$ and $q = r.a$ for some $r \in x$ and $a \in A$.*

For example, all of the non-strict functions in $\mathbb{D}(\mathbf{N} \Rightarrow \mathbf{N})$ that return a result have empty open sets. In contrast, **sub1**, which is $\{\langle ?, 2\rangle.\langle ?, 1\rangle, \langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n+1, 1\rangle.\langle n, 2\rangle | n \in \mathbb{N}\}$, has a non-empty open set, $\{\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.0, 1\rangle\}$.

Definition 6 (Application) *Let $S_1 \Rightarrow S_2$ be the exponent* sds *of S_1 and S_2. The* application *of a tree $t \in \mathbb{D}(S_1 \Rightarrow S_2)$ to some tree $x \in \mathbb{D}(S_1)$, written as $t \star x$, is defined as:*

$$\begin{aligned}
t \star x = \; & \{\pi_2^{\Rightarrow}(p) \mid \|\pi_1^{\Rightarrow}(p)\| \sqsubseteq x, p \in t\}\\
& \cup \{\pi_2^{\Rightarrow}(p).\mathsf{e} \mid \|\pi_1^{\Rightarrow}(p)\| \cup \{q.\mathsf{e}\} \sqsubseteq x, p.\langle q, 1\rangle \in t\}\\
& \cup \{\pi_2^{\Rightarrow}(p).\mathsf{e} \mid \|\pi_1^{\Rightarrow}(p)\| \sqsubseteq x, p.\mathsf{e} \in t\}.
\end{aligned}$$

where $\|\pi_1^{\Rightarrow}(p)\|$ is the domain element that corresponds to the information collected along the path p.

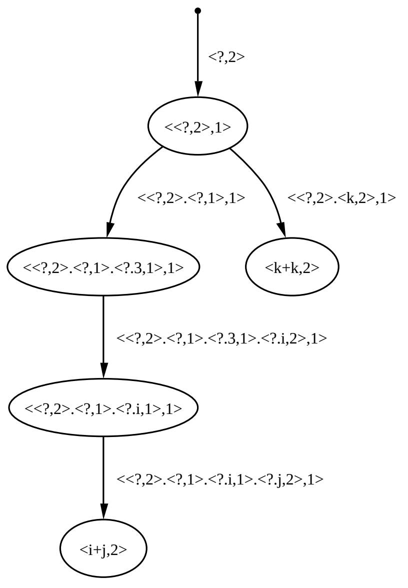


Figure 1. $\lambda f : o \rightarrow o.\ (f\ 3) + (f\ (f\ 3))$ **in** $\mathbb{D}((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N})$

The formal definition of function application just expresses our informal descriptions in the language of paths and domain elements. A function explores the input tree with a sequence of queries and, after collecting enough information, outputs data. The second clause ensures that the function propagates errors from its input if an exploration of the input touches an error. Analogously, if the function contains a path ending in an error, it outputs the error.

Figure 1 introduces a technique for concisely drawing a function tree. The variables i, j, and k in the tree range over $\mathbf{N}$; their use is analogous to set comprehension. The first occurrence (from the root of the tree) specifies a family of paths. For example, i appears as the result of the argument function when applied to 3. Accordingly, the third edge in the left-most path of the tree represents an infinite branch point in the tree, with one path for each natural number. Subsequent uses of the variable refer to the value on that particular path. For example, the final node represents an infinite number of nodes from the earlier infinite branchings; $i + j$ is the result for each instance.

4 SPCF

PCF [24, 27] extends the simply-typed λ-calculus with natural numbers, numeric primitives, and recursion. SPCF [3] extends PCF with errors and exceptions.

Type and Term Syntax

$$
\begin{aligned}
t &= t \to t \mid o \\
M &= c \mid x \mid (\lambda x : t\, .\, M) \mid (M\ M) \mid (f\ M) \\
c &= n \mid \mathsf{ce} \mid \mathsf{e}_1 \mid \mathsf{e}_2 \mid \cdots \\
f &= \mathbf{add1} \mid \mathbf{sub1} \mid \mathbf{if0} \mid \mathbf{Y} \mid \mathbf{catch} \\
n &= 0 \mid 1 \mid \cdots
\end{aligned}
$$

Typing Rules

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

$$\frac{}{\Gamma + \{x : t\} \vdash x : t} \qquad \frac{\Gamma \vdash M : t \to t}{\Gamma \vdash \mathbf{Y}\, M : t} \qquad \frac{\Gamma \vdash \lambda x.M : t}{\Gamma \vdash \mathbf{catch}^t\ (\lambda x.M) : o}$$

$$\frac{}{\Gamma \vdash n : o} \qquad \frac{\Gamma \vdash M : o}{\Gamma \vdash (\mathbf{if0}\ M) : o \to o \to o}$$

$$\frac{\Gamma \vdash M : o}{\Gamma \vdash \mathbf{add1}\ M : o} \qquad \frac{\Gamma \vdash M : o}{\Gamma \vdash \mathbf{sub1}\ M : o} \qquad \frac{}{\Gamma \vdash \mathsf{e} : o}$$

Figure 2. SPCF

4.1 Syntax and Types

Figure 2 specifies SPCF's syntax and type system. The upper half are grammars that specify the raw term and type language. The lower half is dedicated to the typing rules.

The type language of SPCF is that of the simply typed λ-calculus over the natural numbers. Types are either o (natural numbers) or function types built from simpler types.

The PCF term language is basically that of the λ-calculus augmented with **add1** for incrementing natural numbers, **sub1** for decrementing positive numbers, **if0** for branching on natural numbers, and **Y** for creating recursive functions.

SPCF's errors are distinct constants of ground type. Errors stop the evaluation of a program immediately, without recourse. With errors, programmers can observe the sequential evaluation behavior of a program. With **catch**, a program can determine (and exploit) the same sequentiality information. For example, by applying catch to a function f such as $(\lambda i : o.\ \mathbf{if0}\ i\ 0\ (\mathbf{sub1}\ i))$ we can determine that f is strict in its argument and that therefore $(f\ (\mathbf{Y}\ \lambda x.x))$ would diverge and $(f\ \mathsf{e})$ would signal the error e.

The typing rules are the usual ones for the simply typed λ-calculus, plus some rules for the functional constants. The rules use type environments (Γ) and rely on the fact that each variable is superscripted with its type. The most interesting rule is the one for **catch**. It shows that **catch** consumes a function of any type and always produces a o. Later, we omit the types when they can be reconstructed from the context (with unconstrained types treated as o).

$$
\begin{aligned}
&[\![\Gamma \vdash c : t]\!] = [\![c]\!] \circ 1^{1 \times \mathcal{S}eq^{t_n} \times \ldots \times \mathcal{S}eq^{t_1}} \\
&[\![x_n, ..., x_i, ..., x_1 \vdash x_i^t : t]\!] = \pi_i^n \\
&[\![\Gamma \vdash \lambda x^t \,.\, M : t']\!] = \Lambda([\![\Gamma, x^t \vdash M : t']\!]) \qquad x^t \text{ is not in } \Gamma \\
&[\![\Gamma \vdash (M_1\ M_2) : t']\!] = App \circ \langle [\![\Gamma \vdash M_1 : t \to t']\!], [\![\Gamma \vdash M_2 : t]\!] \rangle
\end{aligned}
$$

$[\![M]\!]$ is short for $[\![A \vdash M : t]\!] \star \bot$

$\mathbf{catch}^{o \to o} =$

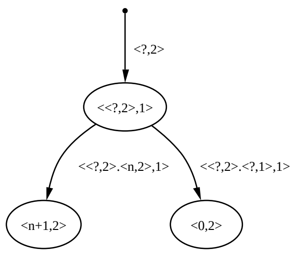


Figure 3. SPCF's mathematical meaning

4.2 Mathematical Meaning

The denotational semantics of SPCF maps a type judgment such as

$$\{x_n^{t_n}, \ldots, x_1^{t_1}\} \vdash M : t$$

to an arrow

$$1 \times \mathcal{S}eq^{t_n} \times \ldots \times \mathcal{S}eq^{t_1} \longrightarrow \mathcal{S}eq^t$$

in $\mathcal{S}eq$. The product on the left represents the shape of the environment in which the term must be evaluated; the object on the right is the interpretation of the term's type.

Figure 3 inductively defines the mapping from type judgments to arrows. The first clause assumes that there is independent definition for the meaning of constants of type t as arrows in $1 \longrightarrow \mathcal{S}eq^t$. For example, the meaning of $\mathbf{catch}^{o \to o}$ is shown in figure 3. For **Y**, we assume the usual limit construction.

4.3 Full Abstraction & Representability

The mathematical semantics is fully abstract [4]. Put differently, for any term M, its tree denotation is a complete representation of how it interacts with all possible contexts in SPCF that turn M into a complete program. As a consequence, M is indistinguishable in SPCF from N iff the two denote equal trees.

Finally, every infinite computable tree in SPCF has a term representation [14]. We can thus easily move back and forth between semantics and syntax, as long as we respect computability. We make use of this fact in the next section where we first explore the candidates for software contracts in $\mathcal{S}eq$ and then their relationship to our syntactic contract system.

5 Contracts as Projections

Equipped with the syntax and semantics of SPCF, we can now study the nature of contracts: what they should be and what they are. We begin this exploration by formulating contracts as mathematical abstractions. Then we analyze each element of our original contract system in this semantic setting, providing a geometric intuition into contracts and then error projections.

5.1 Error Projections

Our first task is to translate the informal discussion of section 2 into formal definitions. Recall that a contract restricts a value or a function's behavior by inserting contract errors. To define this notion of projection, we first define the concept of error approximation.

Definition 7 (Error Approximation) *Let r, r' be two responses from an sds S and let* e *be an error in $\mathbb{E}$. We say $r \sqsubseteq_{\mathsf{e}} r'$ iff*

- *$r = r'$; or*
- *$r = q.\mathsf{e}$ for some $q \in Que_S$ such that $q \sqsubseteq r'$; or*
- *$r = r'.a.\mathsf{e}$ for some address a from S.*

If $x, x' \in \mathbb{D}(S)$ and $\mathsf{e} \in \mathbb{E}$, then $x \sqsubseteq_{\mathsf{e}} x'$ iff

- *for all $r \in x$, there exists an $r' \in x'$ s.t. $r \sqsubseteq_{\mathsf{e}} r'$, and*
- *for all $r' \in x'$, there exists an $r \in x$ s.t. $r \sqsubseteq_{\mathsf{e}} r'$*

Definition 8 (Error Projection) *Let* ce *be the special contract error. A function $p : \mathbb{D}(S) \longrightarrow \mathbb{D}(S)$ is an error projection (written $p \in proj(\mathbb{D}(S))$), if and only if: (1) p is a manifestly sequential function, (2) $p = p \circ p$, and (3) $p(x) \sqsubseteq_{\mathsf{ce}} x$, for all $x \in \mathbb{D}(S)$.*

From now on, we refer to error projections as (semantic) contracts.

5.2 SPCF with Contracts

To support contracts in SPCF, we add one new form: (**contract** M_1 M_2). The first argument is the contract and the second is the value that is monitored. The typing rule:

$$\frac{\Gamma \vdash M_1 : t \rightarrow t \qquad M_2 : t}{\Gamma \vdash (\textbf{contract}\ M_1\ M_2) : t}$$

guarantees that the value and the contract match. Semantically, **contract** applies its first argument to its second.

The key theorem on contracts relates programs to their counterparts where the contracts have been erased.

Definition 9 (Erase) *Erase(M) is like M with all occurrences of (**contract** M_1 M_2) replaced with M_2.*

Theorem 1 (Contract soundness) *For any program M such that the denotation of each first argument to **contract** is an error projection, $[\![M]\!] \sqsubseteq_{\mathsf{ce}} [\![Erase(M)]\!]$.*

Now we can ask how to discharge the antecedent of this theorem, that is, how to construct syntactic contracts so that they always denote semantic contracts. While we had hoped that our original contracts [9] already satisfied that property, in retrospect is is not surprising that they do not.

5.3 Flat Contracts for Flat Values

In λ^{CON}, flat contracts were represented as predicates. Here, we use the FLAT combinator to map predicates to contracts that explore their input with the predicate:

Definition 10 (FLAT)

$$\begin{array}{lcll}
\text{FLAT} & : & \mathbb{D}((C^{t_1} \Rightarrow \cdots C^{t_n}) \Rightarrow \mathbf{N}) \longrightarrow & \\
& & \mathbb{D}(C^{t_1} \Rightarrow \cdots C^{t_n}) \longrightarrow \mathbb{D}(C^{t_1} \Rightarrow \cdots C^{t_n}) & \\
\text{FLAT}_f(x) & = & x & \textit{if } f \star x = \{?.n+1\} \\
& & \{\langle ? \rangle.\mathsf{ce}\} & \textit{if } f \star x = \{?.0\} \\
& & \{\langle ? \rangle.\mathsf{e}\} & \textit{if } f \star x = \{?.\mathsf{e}\} \\
& & \{\} & \textit{if } f \star x = \{\}
\end{array}$$

where $\langle ? \rangle$ is the initial query in $\mathbb{D}(S)$.

Lemma 2 FLAT = $[\![\lambda f : t_1 \rightarrow \cdots (t_n \rightarrow o).\ \lambda x.\ \lambda y_1. \cdots \lambda y_n.$ **if0** $(f\ x)\ \mathsf{ce}\ (x\ y_1 \cdots y_n)]\!]$.

Unfortunately, definition 10 also immediately shows that FLAT cannot produce error projections from all functions in $\mathbb{D}(\mathbf{N}) \rightarrow \mathbb{D}(\mathbf{N})$. The last two clauses in the definition indicate that even a predicate on the natural numbers can have two effects: divergence and errors. Accordingly, we must restrict FLAT's arguments if we wish to construct projections.

Proposition 3 *For any $f : \mathbb{D}(\mathbf{N}) \rightarrow \mathbb{D}(\mathbf{N})$ such that $Open(f) = \emptyset$ and f does not contain any errors, $\text{FLAT}(f) \in proj(\mathbb{D}(M))$.*

5.4 Flat Contracts for Higher-Order Values

Consider the term $\lambda f : o \rightarrow o.\ (f\ \mathsf{0})$. By lemma 2 we know that FLAT $(\lambda f.\ (f\ \mathsf{0}))$ is $\lambda f.\ \lambda x.$ **if0** $(f\ \mathsf{0})\ \mathsf{ce}\ (f\ x)$ in $\mathbb{D}((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}))$. Figure 4 shows the corresponding tree. The longest path in the tree represents the situation when the function with the contract is applied to an integer other than $\mathsf{0}$. This path, however, contains the results of applying the original function to $\mathsf{0}$. Thus, if the original function signals an error for $\mathsf{0}$ but does not for some

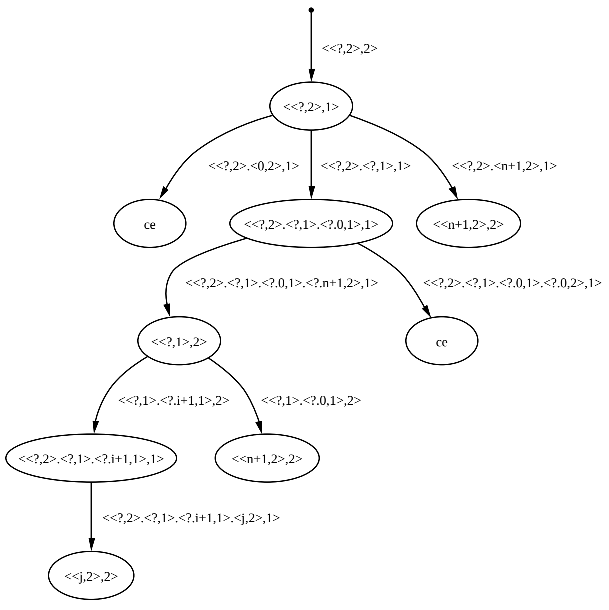


Figure 4. $\lambda f.\ \lambda x.$ **if0** $(f\ 0)$ ce $(f\ x)$

other input value, the contract function signals the error and therefore isn't an error projection. Concretely,

((**contract** ($\lambda f.\ \lambda x.$ **if0** $(f\ 0)$ ce $(f\ x)$)
($\lambda x.$ **if0** x e 33))
1)

is the error e but erasing the contract leaves us with

(($\lambda x.$ **if0** x e 33) 1)

which yields 33. More generally, a predicate on a higher-order domain may not apply its argument if we wish to construct contracts from it.

In $\mathcal{S}eq$, however, predicates can explore functions without applying them. For example, a predicate can test whether a function from **N** to **N** is strict. In SPCF, this action corresponds to applying **catch** to the function. Indeed, testing whether a function probes its argument is a good predicate and yields a contract when supplied to FLAT.

Proposition 4 FLAT(**catch**^{t}) *is an error projection for all t.*

In general, **catch** explores the first-order properties of functions. We refrain from exploring this direction in depth, because it is too closely tied with SPCF's mechanisms of exploring first-class objects.

5.5 Higher-Order Contracts

The idea behind the HO contract combinator is that programmers should be able to restrict the inputs and the outputs of their functions separately. For example, a function of type $o \to o$ should perhaps map odd numbers to prime numbers only. Semantically speaking, we need a function that consumes two error projections and combines them into a new error projection for the matching function type.

Definition 11 (HO**)**

$$
\begin{aligned}
\text{HO}:\ & (\mathbb{D}(C^{t_i}) \to \mathbb{D}(C^{t_i}))\\
&\to (\mathbb{D}(C^{t_o}) \to \mathbb{D}(C^{t_o}))\\
&\to \mathbb{D}(C^{t_i} \Rightarrow C^{t_o}) \to \mathbb{D}(C^{t_i} \Rightarrow C^{t_o})
\end{aligned}
$$

$$
\begin{aligned}
&\text{HO}_{i,o}(f) =\\
&\quad \left\{ \begin{array}{l|l} r \in f & i(||\pi_1^{\Rightarrow}(r)||) = ||\pi_1^{\Rightarrow}(r)|| \\ & o(PC(\pi_2^{\Rightarrow}(r))) = PC(\pi_2^{\Rightarrow}(r)) \end{array} \right\}\\
&\cup\\
&\quad \left\{ \begin{array}{l|l} r.\langle q_i, 1\rangle & r.\langle q_i, 1\rangle \in f \qquad q_i.d_i \in Res_I \\ \ .\langle q_i.d_i, 1\rangle & q_i.\text{ce} \in i \begin{pmatrix} ||\pi_1^{\Rightarrow}(r.\langle q_i, 1\rangle)|| \\ \cup\{q_i.d_i\} \end{pmatrix} \\ \ .\text{ce} & o(PC(\pi_2^{\Rightarrow}(r))) = PC(\pi_2^{\Rightarrow}(r)) \end{array} \right\}\\
&\cup\\
&\quad \left\{ \begin{array}{l|l} q.\text{ce} & q.\langle d_o, 2\rangle \in f \\ & \pi_2^{\Rightarrow}(q).\text{ce} \in o(PC(\pi_2^{\Rightarrow}(q).d_o)) \\ & i(||\pi_1^{\Rightarrow}(q)||) = ||\pi_1^{\Rightarrow}(q)|| \end{array} \right\}
\end{aligned}
$$

where i and o are error projections on the domain and range, $r \in Res_{C^{t_i} \Rightarrow C^{t_o}}$, $q \in Que_{C^{t_i} \Rightarrow C^{t_o}}$, $q_i \in Que_{C^{t_i}}$, d_i is a datum in C^{t_i}, and d_o is a datum in C^{t_o}.

The function $\text{HO}_{i,o}$ uses i and o to process each path in its argument. The processing proceeds according to three cases. Recall that $||\pi_1^{\Rightarrow}(r)||$ is the finite element that the path r inspects before producing its output, and that $\text{PC}(\pi_2^{\Rightarrow}(r))$ is the prefix-closure of the finite element that f produces when the input is approximated by $||\pi_1^{\Rightarrow}(r)||$. The first case deals with paths for which neither i nor o change anything. This means that the function (on that path) keeps its promises and that the input cannot violate its contract. This first situation corresponds to those portions of the constrained function that satisfy both parts of the contract.

The second case deals with paths for which i signals a contract error for the finite portion of the input that the constrained function inspects. Specifically if the function queries the argument with q_i, and there is some datum d_i such that i would insert a contract error when confronted with the path $q_i.d_i$, then the function's original path is replaced by a path that signals the contract error.[3] The third case projects out paths for which o is not the identity. It corresponds to a violation of the range contract.

Lemma 5 HO$=$ $[\![\lambda i\!: t_i \to t_i.\ \lambda o\!: t_o \to t_o.\ \lambda f.\ \lambda x.\ o\ (f\ (i\ x))]\!]$

Even better, $\text{HO}_{i,o}$ always produces error projections from error projections.

[3] It is acceptable if the output projection also cuts its path.

Proposition 6 *For any* $i \in$ *proj*$(\mathbb{D}(I))$ *and* $o \in$ *proj*$(\mathbb{D}(O))$, $\text{HO}_{i,o} \in$ *proj*$(\mathbb{D}(I \Rightarrow O))$.

5.6 Higher-Order Dependent Contracts

Restricting the behavior of a function is not just a matter of restricting its inputs and its outputs separately. Programmers often have a cheap test whether a function produced a reasonable output for some input and need to express this with a function contract. For example, in SPCF (with additional numerics), we can capture *sqrt*'s behavior like this:

(**contract** (HOD (FLAT ($\lambda i.\ i \geq 0$))
    ($\lambda i.$ FLAT ($\lambda o.\ abs(o{*}o - i) \leq 0.001$)))
  sqrt)

This contract states that *sqrt* only accepts numbers bigger than zero and that the result is within 0.001 of the square of the input. The HOD combinator creates this second part of the contract with a function from the input to *sqrt* to a contract for its result.

Definition 12 (HOD)

$$\begin{aligned}\text{HOD}: \ & (\mathbb{D}(C^{t_i}) \to \mathbb{D}(C^{t_i})) \\ & \to (\mathbb{D}(C^{t_i}) \to \mathbb{D}(C^{t_o}) \to \mathbb{D}(C^{t_o})) \\ & \to \mathbb{D}(C^{t_i} \Rightarrow C^{t_o}) \to \mathbb{D}(C^{t_i} \Rightarrow C^{t_o})\end{aligned}$$

$$\text{HOD}_{i,o}(f)(x) = \text{HO}_{i,o(x)}(f)(x)$$

Lemma 7 HOD$=$ $[\![\lambda i\!: t_i \to t_i.\ \lambda o\!: t_i \to t_o \to t_o.\ \lambda f.\ \lambda x.\ (o\ x)\ (f\ (i\ x))]\!]$

To get error projections from HOD, we need to impose stringent constraints on the sub-contract generator that checks the output.

Proposition 8 *For* $i \in$ *proj*$(\mathbb{D}(I))$ *and* $o \in \mathbb{D}(I) \to \mathbb{D}(O) \to \mathbb{D}(O)$ *such that for all* $x \in \mathbb{D}(I)$, $o(x) \in$ *proj*$(\mathbb{D}(O))$*:*

$$\text{HOD}_{i,o} \in proj(\mathbb{D}(I \Rightarrow O)).$$

Unfortunately, this antecedent does not provide a useful criteria for syntactic contracts. Consider this use of HOD:

(HOD (FLAT $\lambda x.$ 1)
    ($\lambda f.$ **if0** (f 0)
        (FLAT $\lambda x.$ 0)
        (FLAT $\lambda x.$ 1))) : $(o \to o) \to (o \to o)$

All of the contracts constructed from FLAT are obviously semantic contracts, because the predicates are total. Hence, the input sub-contract satisfies the antecedent of the proposition. The output sub-contract, however, doesn't. It explores what f produces when given 0 and produces one of two contracts afterwards.

Simplifying the above expression by the equations of the semantics, we have: $\lambda g.\ \lambda f.\ \lambda x.$ **if0** (f 0) ce ((g f) x), since (FLAT $\lambda x.1$) acts as the identity function and (FLAT $\lambda x.0$) rejects all values.

Figure 8 (in the appendix) displays the corresponding domain element. The longest paths in the tree have this shape:

$$\begin{aligned}
&\langle\langle ?, 2\rangle, 2\rangle.\langle\langle\langle ?, 2\rangle, 1\rangle, 2\rangle && (2)\\
&\quad .\langle\langle\langle ?, 2\rangle.\langle ?, 1\rangle, 1\rangle, 2\rangle && (3)\\
&\quad .\langle\langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.0, 1\rangle, 1\rangle, 2\rangle && (4)\\
&\quad .\langle\langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.0, 1\rangle.\langle m+1, 2\rangle, 1\rangle, 2\rangle && (5)\\
&\quad .\langle\langle ?, 1\rangle, 2\rangle && (6)\\
&\quad .\langle\langle ?, 2\rangle.\langle\langle ?, 2\rangle, 1\rangle, 1\rangle && (7)\\
&\quad .\langle\langle ?, 2\rangle.\langle\langle ?, 2\rangle, 1\rangle.\langle\langle ?, 2\rangle.\langle ?, 1\rangle, 1\rangle, 1\rangle && (8)\\
&\quad .\langle\langle ?, 2\rangle && (9)\\
&\quad\ \ .\langle\langle ?, 2\rangle, 1\rangle \\
&\quad\ \ .\langle\langle ?, 2\rangle.\langle ?, 1\rangle, 1\rangle \\
&\quad\ \ .\langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.p+1, 1\rangle, 1\rangle, 1\rangle \\
&\quad .\langle\langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.p+1, 1\rangle, 1\rangle, 2\rangle && (10)\\
&\quad .\langle\langle\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.p+1, 1\rangle.\langle q, 2\rangle, 1\rangle, 2\rangle && (11)\\
&\quad .\langle\langle ?, 2\rangle.\langle\langle ?, 2\rangle, 1\rangle.\langle\langle ?, 2\rangle.\langle q, 2\rangle, 1\rangle, 1\rangle && (12)\\
&\quad .\langle\langle ?, 2\rangle.\langle\langle ?, 2\rangle, 1\rangle.\langle\langle ?, 2\rangle.\langle q, 2\rangle, 1\rangle.\langle r, 2\rangle, 1\rangle && (13)\\
&\quad .\langle\langle r, 2\rangle, 2\rangle && (14)
\end{aligned}$$

This path contains two queries of the input function: step (4) in the path asks for its value at 0 (answered in step (5)) and step (10) asks for its value at $p{+}1$ for some $p \in \mathbb{N}$ (answered in step (11)). Since the context only queries the function at $p+1$, in step (9), this function is not a projection.

Practically put, if the post-condition for HOD explores the argument to the function more than the function itself does, then the antecedent of proposition 8 is guaranteed not to hold. This suggests that we consider an alternative semantic combinator using HO as a template.

Figure 5 contains the alternative definition. It is similar to HO, except that at each place where o is used in HO, HOD$'$ applies o to the finite portion of the input that f already considered along each particular path. This eliminates the above problem, yet still gives the output contract the ability to inspect some portion of the input. Unfortunately, we do not have a topological characterization of the class of contract generators that would enable us to state an analog of proposition 8 for HOD$'$.

Still, we can actually translate this idea into a practical contract combinator. Since the finite elements in $\mathcal{S}eq$ are enumerable, we can hand the contract generator the index of the finite approximation that was explored instead of the element itself. That is, a contract system could use first-order representations of higher-order explorations and thus avoid effectful computations in postconditions.

6 Expressiveness

Since $\mathcal{S}eq$ is fully abstract and since all of its computable elements, including infinite ones, are representable

$$\begin{aligned}\text{HOD}' : \quad & (\mathbb{D}(C^{t_i}) \to \mathbb{D}(C^{t_i})) \\ & \to (\mathbb{D}(C^{t_i}) \to \mathbb{D}(C^{t_o}) \to \mathbb{D}(C^{t_o})) \\ & \to \mathbb{D}(C^{t_i} \Rightarrow C^{t_o}) \to \mathbb{D}(C^{t_i} \Rightarrow C^{t_o})\end{aligned}$$

$$\begin{aligned}
&\text{HOD}'_{i,o}(f) = \\
&\left\{ \begin{array}{l|l} r \in f & i(||\pi_1^{\Rightarrow}(r)||) = ||\pi_1^{\Rightarrow}(r)|| \\ & o(||\pi_1^{\Rightarrow}(r)||)(\text{PC}(\pi_2^{\Rightarrow}(r))) = \text{PC}(\pi_2^{\Rightarrow}(r)) \end{array} \right\} \\
&\cup \\
&\left\{ \begin{array}{l|l} r.\langle q_i, 1\rangle & r.\langle q_i, 1\rangle \in f \qquad q_i.d_i \in \text{Res}_I \\ .\langle q_i.d_i, 1\rangle & q_i.\mathsf{ce} \in i(||\pi_1^{\Rightarrow}(r.\langle q_i, 1\rangle)|| \cup \{q_i.d_i\}) \\ .\mathsf{ce} & o(||\pi_1^{\Rightarrow}(r)||)(\text{PC}(\pi_2^{\Rightarrow}(r))) = \text{PC}(\pi_2^{\Rightarrow}(r)) \end{array} \right\} \\
&\cup \\
&\left\{ \begin{array}{l|l} q.\mathsf{ce} & q.\langle d_o, 2\rangle \in f \\ & \pi_2^{\Rightarrow}(q).\mathsf{ce} \in o(||\pi_1^{\Rightarrow}(q)||)(\text{PC}(\pi_2^{\Rightarrow}(q).d_o)) \\ & i(||\pi_1^{\Rightarrow}(q)||) = ||\pi_1^{\Rightarrow}(q)|| \end{array} \right\}
\end{aligned}$$

Note: $r \in \text{Res}_{C^{t_i} \Rightarrow C^{t_o}}$, $q \in \text{Que}_{C^{t_i} \Rightarrow C^{t_o}}$, $q_i \in \text{Que}_C^{t_i}$, d_i is a datum in C^{t_i}, and d_o is a datum in C^{t_o}.

Figure 5. HOD$'$

in SPCF [4, 14], we know that all semantic contracts are expressible in SPCF. The true question is, however, whether our contract combinators can express all semantic contracts without any additional help.

Theorem 9 *There exist error projections that are not expressible as a combination of* HO, HOD*, and* FLAT.

The proof consists of two lemmas. First, we construct an error projection (using SPCF) on $o \to o$ that explores its function in an unusual manner. Specifically, take a look at the definition of *csid* in figure 6. The function checks two independent properties of it's argument *f*: that *f* is strict and that *f*'s result is the same as its input.

Lemma 10 *The function csid is an error projection.*

The second lemma shows that *csid* is not expressible with the current set of combinators.

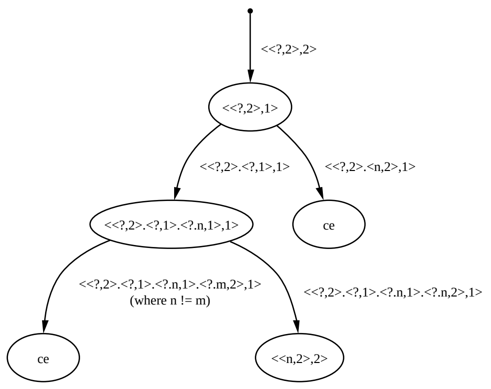
Lemma 11 *The function csid (given in figure 6) is not in the range of* FLAT, HO*, or* HOD.

Extending the language with a new combinator, however, is enough to express *csid*. The new combinator, AND, composes two projections:

Definition 13 $\text{AND}_{f,g}(x) = f(g(x))$

Proposition 12 *csid is*

```
(AND (HOD (FLAT λx. 1)
          (λa. FLAT (λb. (if0 (eq b a) 1 0))))
     (FLAT (λf. (if0 (catch f) 1 0))))
```



```
csid = λf. λx.                    eq = Y (λeq. λx. λy.
         if0 (catch f)                     if0 x y
             (if0 (eq (f x) x)                 (if0 y 1
                  (f x)                             (eq (sub1 x)
                  ce )                                  (sub1 y))))
             ce
```

Figure 6. The contract csid in $\mathbb{D}((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}))$

Two open problems remain. First, we would like to know whether this set of combinators is complete (probably not). Second, if it is not, we would like to find (pragmatic) extensions that make it complete.

7 Blame and Future Work

Our original paper [9] on functional contracts specifies a blame assignment system in addition to an operational semantics of contracts. The blame assignment system identifies the guilty party when a contract violation occurs. For simplicity, it assumes that it takes exactly two parties to agree to a contract: the producer of the value (function, class, object) and the consumer of the value. Technically, the producer is the expression that creates the value; the consumer is the context of this expression.

To distinguish the two possible violations, we can think of each semantic contract to be a composition of two projections. The first monitors the producer's behavior; the second is responsible for the consumer.[4] Adapting the combinators to this new definition is easy; see figure 7 for the result. For flat contracts, the blame lies exclusively with the producer. Hence, the second part of the composition is the identity function. For higher-order contracts, the blame is

[4]In a practical implementation, we also need to add an explanatory message to the respective errors, but we ignore this aspect for now.

$$(\text{FLAT } f) = (\lambda x.\ \lambda y_1 \cdots \lambda y_n.\ \textbf{if0}\ (f\ x)\ \texttt{ce}\ (x\ y_1 \cdots y_n)) \circ \lambda x.\ x$$

$$(\text{HO } (ap \circ an)\ (bp \circ bn)) = \lambda f.\ \lambda x.\ bp\ (f\ (an\ x)) \circ \lambda f.\ \lambda x.\ bn\ (f\ (ap\ x))$$

Figure 7. SPCF/c with Blame

reversed as the contracts is distributed to the domain of a function. Accordingly, the first projection that the HO combinator produces checks the consumer's aspects of the domain and the producer's aspects of the range. Similarly, the second projection checks the producer's aspects of the domain and the consumer's aspects of the range.

The reformulation of the combinators only serves to explicate the nature of blame; their meaning remains the same.

Proposition 13 FLAT *and* HO *in figure 7 are identical to* FLAT *and* HO *in section 5, respectively.*

From that proposition, it follows that the propositions and theorems from the preceding sections apply to this revised contract system as well. In particular, it suffers from all the shortcomings (lack of effect freedom, lack of expressiveness) that the blame-free system has.

The new understanding of contracts has, however, one distinct advantage. It suggest a new ordering relation on error projections. Instead of just comparing contracts with contracts, we can now compare the positive and the negative aspects of a contract with the positive and negative aspects of another contract, respectively. The base comparison (per aspect) is Scott's [29] original ordering relation: $\propto$. To distinguish the two related orderings, we call ours $\lll$.

Definition 14 ($\propto$ and $\lll$)

$a \propto b$ if and only if $a = a \circ b$

$(ap \circ an) \lll (bp \circ bn)$ if and only if $ap \propto bp$ and $bn \propto an$

Intuitively, $a \lll b$ means that a places fewer restrictions on the consumer and more restrictions on the producer of a value than b.

Unlike Scott's original relation, $\lll$ is contra-variant for function arguments and co-variant for function results.

Theorem 14 *For all error projections p, p_1, and p_2 such that $p_1 \lll p_2$,*

$$(\text{HO } p_2\ p) \lll (\text{HO } p_1\ p) \text{ and } (\text{HO } p\ p_1) \lll (\text{HO } p\ p_2).$$

8 Outlook to Software Engineering

In the preceding sections, we have shown that investigating the denotational meaning of contracts provides new and different insights into the area. Thus far, we have had the chance to translate two such insights into practical results.

First, as soon as we noticed that our contract combinators could not express mixtures of first-order and higher-order properties (section 6), we checked whether our Scheme library might benefit from such contracts. Not surprisingly, we found many opportunities. For example, Scheme's *map* accepts a function f and arbitrarily many lists $l_1 \cdots l_n$. Accordingly, f's arity must be n. If so, *map* applies f to each row in the "matrix" of lists, returning a list of the results. If we wish to restrict *map* so that, for example, the lists contain odd integers only, the contract for the restricted *map* has two aspects: a first-order one, which ensures that the arity of f is n, and a higher-order one, which ensures that the functions consume and produce odd integers only.

Second, studying contracts as error projections also pointed out a serious flaw in our previous representation of contracts for Scheme. Specifically, we had distinct representations for first-order and higher-order contracts. As a result, it was extremely difficult to add mixed contracts. Worse, since the contract representation was not uniform, it was nearly impossible to perform optimizations of contracts. This work on contracts as projections pointed out that contracts are always projections (curried, if blame is added as in section 7) and combinator-based combinations of projections. Using this new representation, we improved the performance substantially. For example, checking (HO (FLAT $\lambda x.$ 1) (FLAT $\lambda x.$ 1)) is now 3 times faster than the original system. Better still, adding contracts to our object system became possible; the old representation made adding object contracts intractable.

In addition to these practical improvements, we are now investigating a novel contract language for higher-order objects (closures, objects). This new contract system would use memoization tables to keep track of those portions of an object that a method explores. The contract system would then use this table to conduct run-time checks rather than the object itself. Clearly, this switch would enforce that methods have no visible higher-order effects. The open question is whether such a contract system is practical, and to this end, we will need to conduct experiments.

References

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

[2] 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.

[3] Cartwright, R., P. Curien and M. Felleisen. Observable sequentiality and full abstraction. Technical Report TR-93-219, Rice University, 1993.

[4] Cartwright, R., P. Curien and M. Felleisen. Observable sequentiality and full abstraction. *Information and Computation*, 111(2):297–401, 1994.

[5] Cartwright, R. and M. Fagan. Soft typing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[6] Cheon, Y. A runtime assertion checker for the Java Modelling Language. Technical Report 03-09, Iowa State University Computer Science Department, April 2003.

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

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

[9] Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2002.

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

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

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

[13] Kanneganti, R. and R. Cartwright. What is a universal higher-order programming language? In *International Conference on Automata, Languages and Programming*, pages 682–695, 1993. Springer-Verlag Lecture Notes in Computer Science 700.

[14] Kanneganti, R., R. Cartwright and M. Felleisen. SPCF: its model, calculus, and computational power. In *Sematics: Foundations and Applications, REX Workshop, Beekbergen, The Netherlands*, pages 318–247, 1992. LNCS 666.

[15] 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.

[16] 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.

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

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

[19] Kramer, R. iContract: the Java design by contract tool. In *Technology of OO Languages and Systems*, 1998.

[20] Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek. Report on the programming language Euclid. *ACM Sigplan Notices*, 12(2), Feburary 1977.

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

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

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

[24] Milner, R. Fully abstract models of typed λ-calculi. In *Theoretical Computer Science*, pages 1–22, 1977.

[25] 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.

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

[27] Plotkin, G. D. LCF considered as a programming language. *Theoretical Computer Science*, pages 223–255, 1977.

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

[29] Scott, D. S. Data types as lattices. *Society of Industrial and Applied Mathematics (SIAM) Journal of Computing*, 5(3):522–586, 1976.

Appendix

This appendix consists of figure 8 showing the denotation of $\lambda g.\ \lambda f.\ \lambda x.$ **if0** $(f\ 0)$ ce $((g\ f)\ x)$, and the proofs of the results stated in the main body of the paper.

A Proofs

Theorem 1 *For any program M such that the denotation of each first argument to* **contract** *is an error projection, $[\![M]\!] \sqsubseteq_{\mathsf{ce}} [\![\mathit{Erase}(M)]\!]$.*

Proof. By induction on the structure of M.$\square$

Lemma 2 FLAT = $[\![\lambda x : (t_1 \to \cdots t_n) \to o.\ \lambda y.\ \lambda z_1.\ \cdots \lambda z_n.$ **if0** $(x\ y)$ ce $(y\ z_1 \cdots z_n)\ ]\!]$.

Proof. This proof is four straightforward calculations, one for each case in the definition of FLAT. $\square$

Proposition 3 *For any $f : \mathbb{D}(\mathbf{N}) \to \mathbb{D}(\mathbf{N})$ such that $\mathit{Open}(f) = \emptyset$ and f does not contain any errors,* $\mathrm{FLAT}(f) \in$ *proj*$(\mathbb{D}(M))$.

Proof. The fact that FLAT_f is manifestly sequential follows from an inspection of the definition of FLAT.

To prove that $\mathrm{FLAT}_f = \mathrm{FLAT}_f \circ \mathrm{FLAT}_f$, there are four cases, based on the case distinction in the definition of FLAT. The first case is obvious, as are the third and fourth. For the second case, once we observe that f is manifestly sequential (and thus errors are propagated), the proof goes through. Thus, all uses of FLAT are all retracts.

Due to the constraints on f, we know that the third and fourth cases of FLAT cannot occur unless they come from the argument to FLAT_f and thus FLAT_f satisfies the third condition to be an error projection. $\square$

Proposition 4 $\mathrm{FLAT}(\textbf{catch}^t)$ *is an error projection for all t.*

Proof. For shorthand, we use the name $\mathfrak{S}$ for $\mathrm{FLAT}(\mathrm{catch}^t)$. We have

$$\mathfrak{S}(f) = \begin{cases} \{\langle\langle ?, 2\rangle ..., 2\rangle.\langle\langle \mathsf{ce}, 2\rangle ..., 2\rangle\} \\ \text{ if } \langle\langle ?, 2\rangle ..., 2\rangle.\langle\langle ?, 1\rangle, 2\rangle ..., 2\rangle \in f \\ \\ f \text{ otherwise} \end{cases}$$

This function is clearly manifestly sequential and also clearly a retract. To show that it is an error projection, let f be given and consider $\mathfrak{S}(f)$. If f falls into the second case of $\mathfrak{S}$, then clearly $\mathfrak{S}(f) \sqsubseteq_{\mathsf{ce}} f$.

If f falls into the first case, the only element of $\mathfrak{S}(f)$ is as given above. Clearly, that element is an error approximation to an element of f, by the condition for being in the first case. Also, since the path given in the condition for the first case is of length two, it is the shortest path in f and therefore the only element of $\mathfrak{S}(f)$ is an error approximation of all elements of f. $\square$

Lemma 5 HO= $[\![\lambda i\!: t_i \to t_i.\ \lambda o\!: t_o \to t_o.\ \lambda f.\ \lambda x.\ o\ (f\ (i\ x))]\!]$

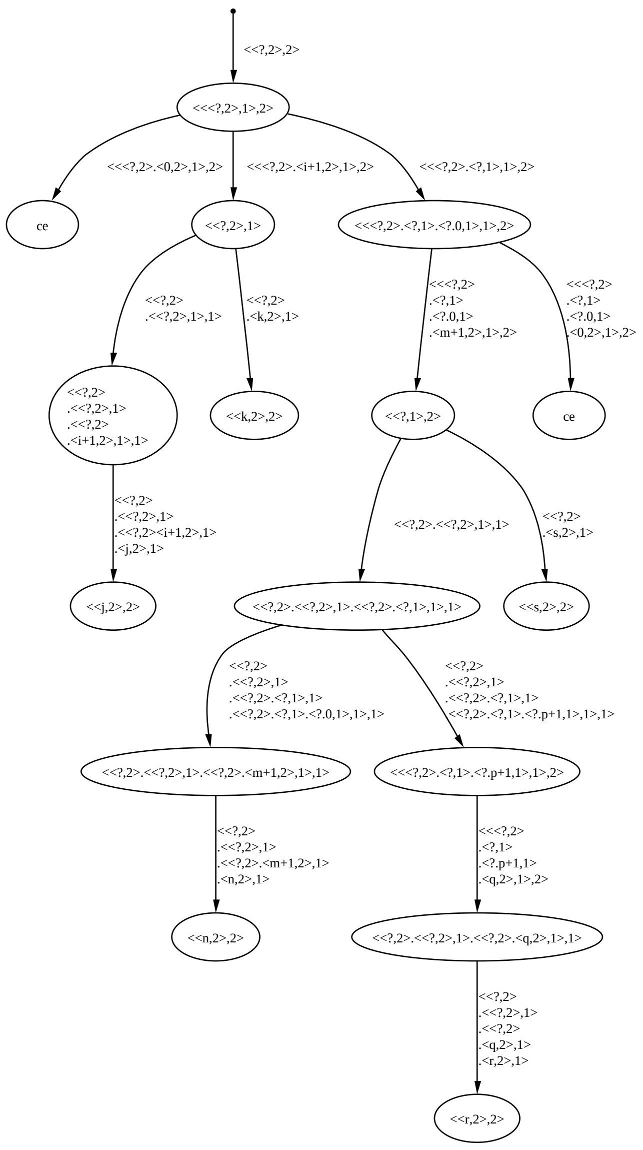

<<?,2>,2>
<<<?,2>,1>,2>
<<<?,2>.<0,2>,1>,2>
<<<?,2>.<i+1,2>,1>,2>
<<<?,2>.<?,1>,1>,2>
ce
<<?,2>,1>
<<<?,2>.<?,1>.<?.0,1>,1>,2>
<<?,2> .<<?,2>,1>,1>
<<?,2> .<k,2>,1>
<<<?,2> .<?,1> .<?.0,1> .<m+1,2>,1>,2>
<<<?,2> .<?,1> .<?.0,1> .<0,2>,1>,2>
<<?,2> .<<?,2>,1> .<<?,2> .<i+1,2>,1>,1>
<<k,2>,2>
<<?,1>,2>
ce
<<?,2> .<<?,2>,1> .<<?,2><i+1,2>,1> .<j,2>,1>
<<?,2>.<<?,2>,1>,1>
<<?,2> .<s,2>,1>
<<j,2>,2>
<<?,2>.<<?,2>,1>.<<?,2>.<?,1>,1>,1>
<<s,2>,2>
<<?,2> .<<?,2>,1> .<<?,2>.<?,1>,1> .<<?,2>.<?,1>.<?.0,1>,1>,1>
<<?,2> .<<?,2>,1> .<<?,2>.<?,1>,1> <<?,2>.<?,1>.<?.p+1,1>,1>,1>
<<?,2>.<<?,2>,1>.<<?,2>.<m+1,2>,1>,1>
<<<?,2>.<?,1>.<?.p+1,1>,1>,2>
<<?,2> .<<?,2>,1> .<<?,2>.<m+1,2>,1> .<n,2>,1>
<<<?,2> .<?,1> .<?.p+1,1> .<q,2>,1>,2>
<<n,2>,2>
<<?,2>.<<?,2>,1>.<<?,2>.<q,2>,1>,1>
<<?,2> .<<?,2>,1> .<<?,2> .<q,2>,1> .<r,2>,1>
<<r,2>,2>

Figure 8. The tree for $\lambda g.\ \lambda f.\ \lambda x.$ **if0** $(f\ 0)$ ce $((g\ f)\ x)$ **in** $\mathbb{D}(((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N})$

Proof. By extensionality (theorem 6.12 [4]) and lemma 15. $\square$

Proposition 6 *For any $i \in$ proj$(\mathbb{D}(I))$ and $o \in$ proj$(\mathbb{D}(O))$,* $\mathrm{HO}_{i,o} \in$ *proj*$(\mathbb{D}(I \Rightarrow O))$.

Proof. The definition of HO shows that it is manifestly sequential. Similarly, it also implies $\mathrm{HO}_{i,o}(f) \sqsubseteq_{\mathsf{ce}} f$ for any

$f \in \mathbb{D}(I \Rightarrow O)$. The idempotency of HO follows from a case analysis on the three cases in its definition. □

Lemma 7 HOD= $[\![\lambda i\colon t_i \to t_i.\ \lambda o\colon t_i \to t_o \to t_o.\ \lambda f.\ \lambda x.\ (o\ x)\ (f\ (i\ x))]\!]$

Proof. Follows from the definition of HOD and lemma 5. □

Proposition 8 *For $i \in proj(\mathbb{D}(I))$ and $o \in \mathbb{D}(I) \to \mathbb{D}(O) \to \mathbb{D}(O)$ such that for all $x \in \mathbb{D}(I)$, $o(x) \in proj(\mathbb{D}(O))$:*

$$\text{HOD}_{i,o} \in proj(\mathbb{D}(I \Rightarrow O)).$$

Proof. The proof is analogous to that of proposition 6. □

Lemma 10 *The function csid is an error projection.*

Proof. Let f be in $\mathbb{D}(\mathbf{N}) \longrightarrow \mathbb{D}(\mathbf{N})$. The function csid produces one of the following values for such an f:

$$\begin{array}{ll}
\{\langle ?, 2\rangle.\mathsf{ce}\} & \text{if } \langle ?, 2\rangle.\langle n, 2\rangle \in f \\
\left.\begin{array}{l}
\quad \{\langle ?, 2\rangle.\langle ?, 1\rangle\} \\
\cup \{\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n, 1\rangle.\mathsf{ce} \mid \\
\quad \langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n, 1\rangle.\langle m, 2\rangle \in f, \\
\quad n \neq m\} \\
\cup \{\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n, 1\rangle.\langle n, 2\rangle \mid \\
\quad \langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n, 1\rangle.\langle n, 2\rangle \in f\}
\end{array}\right\} & \text{otherwise}
\end{array}$$

This follows from the denotation (see the top of figure 6) of *csid*. Clearly, *csid* is a retract and, for all f, (*csid* f) error approximates f. □

Lemma 11 *The function csid (given in figure 6) is not in the range of* FLAT, HO, *or* HOD.

Proof. To show that *csid* is not in the range of HOD, first assume that it is. Thus, we can pick i, o such that $\text{HOD}_{i,o} = csid$. We know that $[\![(csid\ (\lambda x.\ 3))]\!] = \{\langle ?.2\rangle.\mathsf{ce}\}$. Since $[\![\lambda x.\ 3]\!] = \{\langle ?, 2\rangle.\langle 3, 2\rangle\}$, $\text{HOD}_{i,o}(\{\langle ?, 2\rangle.\langle 3, 2\rangle\}) = \{\langle ?.2\rangle.\mathsf{ce}\}$ and applying that to bottom gives $\{?.\mathsf{ce}\}$. Since ce does not appear in the argument to $\text{HOD}_{i,o}$, it must be introduced in either the second or third subset in the definition of HO. It cannot have been introduced in the second subset, since there are no paths of the form $p.\langle q, 1\rangle$ in HOD's argument. Thus, we know that it comes from the third subset and $o(\{\})(\{?.3\}) = \{?.\mathsf{ce}\}$.

Now consider the element:

$$\begin{aligned} f &\overset{\text{def}}{=} \{\langle ?, 2\rangle.\langle ?, 1\rangle, \langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.n, 1\rangle.\langle 3, 2\rangle | n \in \mathbb{N}\} \\ &= [\![\lambda x.\ \mathbf{if0}\ x\ 3\ 3]\!] \end{aligned}$$

We know that ((*csid* ($\lambda x.$ **if0** x 3 3)) 3) = 3. Accordingly, $\text{HOD}_{i,o}(f)(\{?.3\}) = \{?.3\}$.

Since o must be monotonic, we know that $o(\{?.3\})(\{?.3\})$ must be $\{?.\mathsf{ce}\}$. Considering the third set in the definition of HO again, if $\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.3, 1\rangle.ce$ is an element of $\text{HOD}_{i,o}(f)$, then $\text{HOD}_{i,o}(f) \star \{?.3\} = \{?.\mathsf{ce}\}$. So, it must be the case that $\langle ?, 2\rangle.\langle ?, 1\rangle.\langle ?.3, 1\rangle.ce$ is not in $\text{HOD}_{i,o}(f) \star \{?.3\}$. Since we have already established o's behavior for this element, in order to rule out that path, we know that $i(\{?.3\}) \neq (\{?.3\})$. Since i is an error projection, it must be the case that $i(\{?.3\}) = \{?.\mathsf{ce}\}$. But that would also mean that $\text{HOD}_{i,o}(f) \star \{?.3\} = \{?.\mathsf{ce}\}$, which is a contradiction. Therefore *csid* is not in the range of HOD. Since every function in the range of HO is also in the range of HOD, *csid* is not in the range of HO. (A similar argument to the above shows that *csid* is not in the range of HOD$'$.)

To show that *csid* is not in the range of FLAT, first assume that it is. Thus, we can pick p such that $csid = \lambda f.$ **if0** (p f) f ce. For each natural number i, define $f_i : \mathbb{D}(\mathbf{N}) \to \mathbb{D}(\mathbf{N})$

$$f_i(x) = \begin{cases} \{\} & \text{if } x = \{\} \\ \{?.x\} & \text{if } x = \{?.n\} \text{ and } 0 < n \leq i \\ \{\} & \text{if } x = \{?.n\} \text{ and } n > i \end{cases}$$

Define $f = \bigsqcup_i f_i$ and $g : \mathbb{D}(\mathbf{N}) \to \mathbb{D}(\mathbf{N})$

$$g(x) = \begin{cases} \{\} & \text{if } x = \{\} \\ \{?.1\} & \text{if } x = \{?.0\} \\ \{?.n\} & \text{if } x = \{?.n\} \text{ and } n \neq 0 \end{cases}$$

We know that $([\![csid]\!] \star g) \star \{?.0\} = \{?.\mathsf{ce}\}$ and thus $[\![p]\!] \star g = \{?.n+1\}$. Also, we know that since $([\![csid]\!] \star f_i) \star \{?.i\} \sqsubseteq \{?.i\}$, $[\![p]\!] \star f_i = \{?.0\}$ for any i. But this leads to a contradiction:

$$\begin{aligned}
\{?.0\} &= \textstyle\bigsqcup_i \{\{?.0\}\} && \text{by definition of } \textstyle\bigsqcup \\
&= \textstyle\bigsqcup_i \{[\![p]\!] \star f_i\} && \text{by above observation} \\
&= [\![p]\!] \star (\textstyle\bigsqcup_i \{f_i\}) && \text{by continuity of } [\![p]\!] \\
&= [\![p]\!] \star f && \text{by definition of } f \\
&\sqsubseteq [\![p]\!] \star g && \text{by monotonicity of } [\![p]\!] \text{ and } f \sqsubseteq g \\
&= \{?.n+1\} && \text{by above observation}
\end{aligned}$$

because $\{?.0\} \not\sqsubseteq \{?.n+1\}$. Therefore *csid* is not in the range of FLAT. □

Theorem 12 *csid is*

*(*AND *(*HOD *(*FLAT *λx.* 1*)*
(λa. FLAT *(λb. (***if0** *(eq b a)* 1 0*))))*
*(*FLAT *(λf. (***if0** *(***catch** *f)* 1 0*))))*

Proof. The proof is a somewhat lengthy calculation involving standard equations of SPCF, lemmas 2 and 7 showing HOD and FLAT's behavior, and these additional equations:

if0 swapping: (**if** (**if0** a 1 0) b c) = (**if0** a c b)

if0 factoring: (**if0** (**if0** a b c) d e) = (**if0** a (**if0** b d e) (**if0** c d e))

if0 nesting: (**if0** x y D[(**if0** x z w)]) = (**if0** x y $D[w]$)

$$\begin{array}{lrl} & D \;= & (\mathbf{if0}\ D\ M\ M) \\ \text{where} & | & (\mathbf{if0}\ M\ D\ M) \\ & | & (\mathbf{if0}\ M\ M\ D) \\ & | & E \end{array}$$

□

Lemma 15 *For any* $x \in \mathbb{D}(I), f \in \mathbb{D}(I \Rightarrow O), i \in$ *proj*$(\mathbb{D}(I)), o \in$ *proj*$(\mathbb{D}(O))$,

$$\text{HO}_{i,o}(f) \star x = o(f \star i(x))$$

Proof. Let $r \in \text{Res}^{\mathbb{E}}_O$ be given.

▶ **Case** $\subseteq$**.** Assume $r \in \text{HO}_{i,o}(f) \star x$.

▶ **Case** $\subseteq$**1.** Assume that r is the first subset of the definition of $\star$. Pick p such that $r = \pi^{\Rightarrow}_2(p), r \in \text{Res}_O, ||\pi^{\Rightarrow}_1(p)|| \sqsubseteq x$, and $p \in \text{HO}_{i,o}(f)$. Since there are no errors in the first subset of $\star$, there are no errors in r and we know that p must come from the first subset of $\text{HO}_{i,o}(f)$. Thus, $i(||\pi^{\Rightarrow}_1(p)||) = ||\pi^{\Rightarrow}_1(p)||, o(\text{PC}(\pi^{\Rightarrow}_2(p))) = \text{PC}(\pi^{\Rightarrow}_2(p))$, and $p \in f$. Since i is monotone, we know $i(||\pi^{\Rightarrow}_1(p)||) \sqsubseteq i(x)$ and thus $||\pi^{\Rightarrow}_1(p)|| \sqsubseteq i(x)$. Accordingly, by the definition of $\star$, $p \in f \star i(x)$. Thus $\text{PC}(\pi^{\Rightarrow}_2(p)) \sqsubseteq f \star i(x)$ and by the monotonicity of o, $o(\text{PC}(\pi^{\Rightarrow}_2(p))) \sqsubseteq o(f \star i(x))$. Finally, since $o(\text{PC}(\pi^{\Rightarrow}_2(p))) = \text{PC}(\pi^{\Rightarrow}_2(p))$ and $r \in \text{PC}(\pi^{\Rightarrow}_2(p)), r \in o(f \star i(x))$.

▶ **Case** $\subseteq$**2.** Assume that r is in the second subset of the definition of $\star$. Pick p, q such that $r = \pi^{\Rightarrow}_2(p).\mathsf{e}, ||\pi^{\Rightarrow}_1(p)|| \cup \{q.\mathsf{e}\} \sqsubseteq x$, and $p.\langle q, 1\rangle \in \text{HO}_{i,o}(f)$. Since $p.\langle q, 1\rangle$ does not end in an error, we know that it must come from the first case of $\text{HO}_{i,o}(f)$. Accordingly, $i(||\pi^{\Rightarrow}_1(p.\langle q, 1\rangle)) = ||\pi^{\Rightarrow}_1(p.\langle q, 1\rangle)||$ and $o(\text{PC}(\pi^{\Rightarrow}_2(p))) = \text{PC}(\pi^{\Rightarrow}_2(p))$. Since i is manifestly sequential, $i(\{q.\mathsf{e}\}) = \{q.\mathsf{e}\}$. Thus, $i(||\pi^{\Rightarrow}_1(p)|| \cup \{q.\mathsf{e}\}) = ||\pi^{\Rightarrow}_1(p)|| \cup \{q.\mathsf{e}\}$ and $||\pi^{\Rightarrow}_1(p)|| \cup \{q.\mathsf{e}\} \sqsubseteq i(x)$. Accordingly, $\pi^{\Rightarrow}_2(p).\mathsf{e} \in f \star i(x)$. Since o is monotonic, $o(\text{PC}(\pi^{\Rightarrow}_2(p).\mathsf{e})) \sqsubseteq o(f \star i(x))$. Since o is manifestly sequential $o(\text{PC}(\pi^{\Rightarrow}_2(p).\mathsf{e})) = \text{PC}(\pi^{\Rightarrow}_2(p).\mathsf{e})$ and $\pi^{\Rightarrow}_2(p).\mathsf{e} \in (f \star i(x))$.

▶ **Case** $\subseteq$3. Assume that r is in the third subset of the definition of $\star$. Choose p such that $r = \pi^{\Rightarrow}_2(p).\mathsf{e}, p.\mathsf{e} \in \text{HO}_{i,o}(f)$, and $||\pi^{\Rightarrow}_1(p)|| \sqsubseteq x$.

▶ **Case** $\subseteq$**3a.** Assume that $p.\mathsf{e}$ comes from the first case in the definition of $\text{HO}_{i,o}(f)$. Thus, $o(\text{PC}(\pi^{\Rightarrow}_2(p.\mathsf{e}))) = \text{PC}(\pi^{\Rightarrow}_2(p.\mathsf{e}))$ and $i(||\pi^{\Rightarrow}_1(p.\mathsf{e})||) = ||\pi^{\Rightarrow}_1(p.\mathsf{e})||$. As before, $||\pi^{\Rightarrow}_1(p.\mathsf{e})|| = i(||\pi^{\Rightarrow}_1(p.\mathsf{e}||) \sqsubseteq i(x)$. Thus, $\pi^{\Rightarrow}_2(p).\mathsf{e} \in f \star i(x)$. Also as before, $\pi^{\Rightarrow}_2(p).\mathsf{e} \in \text{PC}(\pi^{\Rightarrow}_2(p).\mathsf{e} = o(\text{PC}(\pi^{\Rightarrow}_2(p.\mathsf{e}))) \sqsubseteq o(f \star i(x))$.

▶ **Case** $\subseteq$**3b.** Assume that $p.\mathsf{e}$ comes from the second case in the definition of $\text{HO}_{i,o}(f)$. Thus, $\mathsf{e} = \mathsf{ce}, p = r.\langle q_i, d_i, 1\rangle.\mathsf{ce}$, and $i(||\pi^{\Rightarrow}_1(r.\langle q_i.d_i, 1\rangle||) \sqsubseteq i(x)$. Since $q_i.\mathsf{ce} \in ||\pi^{\Rightarrow}_1(r.\langle q_i.d_i, 1\rangle)||$, we know that $q_i.\mathsf{ce} \in i(x)$. According to the definition of $\star$, $r \in f \star i(x)$. Thus, as in the previous cases, $r \in o(f \star i(x))$.

▶ **Case** $\subseteq$**3c.** Assume that $p.\mathsf{e}$ comes from the third case in the definition of $\text{HO}_{i,o}(f)$. Thus, $\mathsf{e} = \mathsf{ce}$. Pick q, d_o such that $q.\langle d_o, 2\rangle \in f, \pi^{\Rightarrow}_2(q).\mathsf{ce} \in o(\text{PC}(\pi^{\Rightarrow}_2(q).d_o))$, and $i(||\pi^{\Rightarrow}_1(q)||) = ||\pi^{\Rightarrow}_1(q)||$. As in previous cases, by the monotonicity of i, we know that $||\pi^{\Rightarrow}_1(p.\langle d_o, 2\rangle)|| \sqsubseteq i(x)$. Thus, $\pi^{\Rightarrow}_2(p.\langle d_o, 2\rangle) \in f \star i(x)$. Since o is monotonic, $o(\text{PC}(\pi^{\Rightarrow}_2(p.\langle d_o, 2\rangle))) \in o(f \star i(x))$. Thus, since $\pi^{\Rightarrow}_2(p).\mathsf{ce} \in o(\text{PC}(\pi^{\Rightarrow}_2(p.\langle d_o, 2\rangle)))$, $r \in o(f \star i(o))$.

Therefore, $\text{HO}_{i,o}(f) \star x \subseteq o(f \star i(x))$.

▶ **Case** $\supseteq$**.** Assume $r \in o(f \star i(x))$.

▶ **Case** $\supseteq$1. Assume that $r \in f \star i(x)$ but $r \notin o(f \star i(x))$. Then, there exists $rl \in f \star i(x), qn \in \text{Que}_O$ such that $r \sqsubset rl, o(rl) = r$, and $r = qn.\mathsf{e}$. If $rl \in \text{Res}_O$, let rs be rl. Otherwise, since rl does end in an error, $rl = qm.\mathsf{e}$ for some $qm \in \text{Que}_O$. Since o is manifestly sequential and $rl \notin o(\text{PC}(rl))$, we know that $qn \sqsubset qm$. Thus, there exists $rs \in \text{Res}_O$ such that $qn \sqsubset rs \sqsubset qm$ and $rs \in f \star i(x)$.

From the definition of $\star$, choose p such that $\pi^{\Rightarrow}_2(p) = rs, p \in f, p = q.\langle d_o, 2\rangle$, and $||\pi^{\Rightarrow}_1(p)|| \sqsubseteq i(x)$. Assume that $||\pi^{\Rightarrow}_1(p)|| \not\sqsubseteq x$. By the definition of error projection, there exists a q such that $q.\mathsf{ce} \in ||\pi^{\Rightarrow}_1(p)||$. But, $\pi^{\Rightarrow}_1(p)$ for any p never contains an error, so $||\pi^{\Rightarrow}_1(p)|| \sqsubseteq x$. Accordingly, $||\pi^{\Rightarrow}_1(p)||$ is in the range of i and $||\pi^{\Rightarrow}_1(p)|| = i(||\pi^{\Rightarrow}_1(p)||)$. Thus, $q.\mathsf{ce} \in \text{HO}_{i,o}(f)$, from the second case of the definition of $\text{HO}_{i,o}$. Since $||\pi^{\Rightarrow}_1(q.\mathsf{ce})|| \sqsubseteq x$, $\pi^{\Rightarrow}_2(q).\mathsf{ce} \in \text{HO}_{i,o}(f) \star x$ Since $r \sqsubset rs$, we know that $r \sqsubseteq \pi^{\Rightarrow}_2(q).ce$ and thus $r \in \text{HO}_{i,o}(f) \star x$.

▶ **Case** $\supseteq$2. Assume that $r \in f \star i(x)$ and $r \in o(f \star i(x))$. Thus $o(\text{PC}(r)) = \text{PC}(r)$.

▶ **Case** $\supseteq$2a. Assume $r \in \text{Res}_O$. That is, r does not end in an error. From the definition of $\star$, we can pick p such that $\pi^{\Rightarrow}_2(p) = r, ||\pi^{\Rightarrow}_1(p)|| \sqsubset i(x)$, and $p \in f$. Since $||\pi^{\Rightarrow}_1(p)||$ has no error, we know that $||\pi^{\Rightarrow}_1(p)|| \sqsubset x$. Thus, $p \in \text{HO}_{i,o}(f)$. From the definition of $\star$, $r \in \text{HO}_{i,o}(f) \star x$.

▶ **Case** $\supseteq$2b. $r = qn.\mathsf{e}$ and r comes from the second set of the definition of $\star$. Choose p, q such that $||\pi^{\Rightarrow}_1(p)|| \cup \{q.\mathsf{e}\} \sqsubseteq i(x), \pi^{\Rightarrow}_2(p) = r$, and $p.\langle q, 1\rangle \in f$. If $i(||\pi^{\Rightarrow}_1(p)||) = ||\pi^{\Rightarrow}_1(p)||$, take p' to be p and q' to be q. Otherwise, we can construct p' such that $p'.\langle q', 1\rangle \sqsubset p$ and $i(||\pi^{\Rightarrow}_1(p')||) = ||\pi^{\Rightarrow}_1(p)$ by trimming p to the first place where i inserts an error in the first projection of p.

If $||\pi^{\Rightarrow}_1(p')|| \cup \{q'.e\} \sqsubseteq x$, then $i(||\pi^{\Rightarrow}_1(p')|| \cup \{q'.e\}) = ||\pi^{\Rightarrow}_1(p')|| \cup \{q'.e\}$ and $p' \in \text{HO}_{i,o}(f)$. Thus $r \in \text{HO}_{i,o}(f) \star x$. If not, $\mathsf{e} = \mathsf{ce}$ and $q'.\mathsf{ce} \in i(\text{PC}(q'.d))$ where d is some datum of I and $q'.d \in x$. Accordingly, $p'.\langle q', 1\rangle.\langle q'.d', 1\rangle.\mathsf{ce} \in \text{HO}_{i,o}(f)$ and $||\pi^{\Rightarrow}_1(p'.\langle q', 1\rangle.\langle q'.d, 1\rangle)|| \sqsubset x$. Thus, $\pi^{\Rightarrow}_2(p.\langle q, 1\rangle.\langle q.d, 1\rangle).\mathsf{ce} \in \text{HO}_{i,o}(f) \star x$ and therefore $r \in \text{HO}_{i,o}(f) \star x$.

▶ **Case** $\supseteq$2c. $r = rn.e$ and comes from the third set in the definition of $\star$. Pick p such that $p.\mathsf{e} \in f$ and $||\pi^{\Rightarrow}_1(p)|| \sqsubseteq i(x)$. We know that $||\pi^{\Rightarrow}_1(p)|| \sqsubseteq x$ since $||\pi^{\Rightarrow}_1(p)||$ has no errors. Thus, $p.e \in \text{HO}_{i,o}(f)$ and $r \in \text{HO}_{i,o}(f) \star x$.

Therefore, $\text{HO}_{i,o}(f) \star x \supseteq o(f \star i(x))$, concluding the proof. □

Proposition 13 FLAT *and* HO *in figure 7 are identical to* FLAT *and* HO *in section 5, respectively.*

Proof. For FLAT, this is obvious, and for HO a simple calculation, using only β, proves the result. □

Theorem 14 *For all error projections* p, p_1*, and* p_2 *such that* $p_1 \lll p_2$,

$$(\textsc{ho}\ p_2\ p) \lll (\textsc{ho}\ p_1\ p) \textit{ and } (\textsc{ho}\ p\ p_1) \lll (\textsc{ho}\ p\ p_2).$$

Proof. This is a slightly longer calculation than the previous proof and relies on the idempotence of error projections, β, and the fact that $p_1 \circ p_2 = p_2 \circ p_1$ when p_1 and p_2 are both error projections that only signal the same error. Equipped with these equations, however, the theorem follows directly from the definitions. □