SPCF: Its Model, Calculus, and Computational Power (Preliminary Version)

Ramarao Kanneganti, Robert Cartwright, Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251-1892

A version of this report appears in the
Proceedings of the 1992 REX Workshop

Contents

SPCF: Its Model, Calculus, and Computational Power (Preliminary Version)

Ramarao Kanneganti, Robert Cartwright, Matthias Felleisen*
Department of Computer Science
Rice University
Houston, TX 77005

Abstract

SPCF is an idealized sequential programming language, based on Plotkin's language PCF, that permits programmers and programs to observe the evaluation order of procedures. In this paper, we construct a fully abstract model of SPCF using a new mathematical framework suitable for defining fully abstract models of sequential functional languages. Then, we develop an extended typed λ-calculus to specify the operational semantics of SPCF and show that the calculus is complete for the constant-free sublanguage. Finally, we prove that SPCF is *computationally complete*: it can express all the computable (recursively enumerable) elements in its fully abstract model.

1 SPCF: Observing Sequentiality

Most contemporary programming languages, *e.g.*, Scheme, Pascal, Fortran, C, and ML, are "sequential", that is, they impose a serial order on the evaluation of parts of programs. Unfortunately, the familiar mathematical models for sequential languages based on continuous functions do not capture this property. They make artificial distinctions between phrases with the same sequential behavior, preventing the familiar models from being *fully abstract*. Milner [19] and Plotkin [22] recognized this problem nearly twenty years ago and identified the construction of sequential denotational language models as an important research problem.

Early work in the search for "sequential models" focused on the typed λ-calculus with constants for arithmetic and recursion (PCF) [32, 19, 22, 17, 3, 4, 6, 11, 20]. While this strategy avoids many arbitrary language design decisions and simplifies the investigation, it also eliminates programming facilities that are essential for understanding the sequential behavior of programs. The missing facilities include error values, which permit a programmer to observe the order of evaluation among sub-expressions, and constructs for non-local transfer of control, which permit a program to detect and exploit the evaluation order of procedures.

*The authors were supported in part by NSF grants CCR 89-17022 and CCR 91-22518.

Cartwright and Felleisen [9] recently observed that the addition of error values and control constructs to the idealized programming language PCF facilitated the construction of a natural, fully abstract denotational model. Most importantly, the same construction is applicable to variations of PCF with call-by-value parameter-passing and with constants for observing the behavior of higher-order expressions as long as these languages include error values and appropriate control constructs. Instead of continuous functions, the Cartwright-Felleisen model uses *decision trees* to assign meaning to procedures. In a subsequent paper, Curien [12] showed that the decision tree model is an extension of the sequential algorithm model [4, 11], and that the set of observably sequential domains and functions form a Cartesian-closed category.

In this paper, we continue the investigation of SPCF, Cartwright and Felleisen's extension of PCF. First, we reconstruct the model using simpler, more abstract methods based on Winskel's [33] idea of a prime basis (also compare Kahn and Plokin's technical report on concrete domains Second, we show that the extended language corresponds to a simple extension of the typed λ-calculus and prove that the constant-free fragment of the calculus is complete. Finally, we prove that SPCF is computationally complete, that is, it can express all computable (recursively enumerable) elements of the model, including those of higher-order function domains.

2 From PCF to SPCF

PCF is an extension of the simply typed λ-calculus. In addition to the usual λ-calculus constructs, it provides a family of fixpoint operators and a set of constant symbols for performing arithmetic (non-negative numerals, a successor, a predecessor, and a zero recognizer). SPCF enriches PCF in two ways. First, it adds error values to the data domain so that the behavior of a program reveals when functions are *misapplied* to arguments, *e.g.*, an attempt to subtract from zero or to divide by zero. Second, SPCF includes a family of control constructs for escaping from the evaluation of a phrase. In PCF, both of these capabilities are missing. Programs that misapply functions merely diverge. PCF does not include any non-local control constructs, so programs cannot escape from pending computations.

The first part of Figure 1 defines the set of syntactically well-formed SPCF phrases. An SPCF phrase M is either a constant (i.e., a numeral $\ulcorner n \urcorner$, $n \geq 0$, one of two error constants, error_1 and error_2, or a functional constant), a typed variable x^τ, a λ-abstraction, or an application. We call an application of the form ($\mathsf{call/cdc}\ \lambda x.M$) a $\mathsf{call/cdc}$-*application*; the variable x is called a *downward continuation* (*variable*).

As usual, λ-abstraction is the only binding form in this language: $\lambda x.M$ binds x in M. A variable that is not bound by some surrounding λ-abstraction is *free*. We identify phrases that are equal up to renaming bound variables, *e.g.*, $\lambda x.x \equiv \lambda y.y$. We use the standard syntactic conventions of the λ-calculus for writing SPCF phrases, e.g., $\lambda xyz.M$ is short for $\lambda x.(\lambda y.(\lambda z.M))$. In the remainder of the paper, we will use the symbol $\mathcal{V}$ to denote the set of SPCF variable names.

A typed SPCF phrase is an untyped phrase that conforms to the typing rules of SPCF given in Figure 1. The set of SPCF types consists of a single ground type (o), denoting the observable set of non-negative integers, and an infinite collection of finitely generated

Syntax:

$$\begin{array}{rcl} M & ::= & c \mid x \mid (\lambda x.M) \mid (M\ M) \\ c & ::= & \ulcorner n \urcorner \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \mathsf{add1} \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{Y}^\tau \mid \mathsf{call/cdc} \\ x & ::= & x^\tau \mid y^\tau \mid \ldots \end{array}$$

Types: $\sigma, \tau ::= o \mid (\tau \to \sigma)$

Type Checking:

$$\frac{A, x^\tau \vdash M : \tau'}{A \vdash \lambda x^\tau.M : \tau \to \tau'} \qquad \frac{A \vdash M : \tau' \to \tau; \quad A \vdash M' : \tau'}{A \vdash (MM') : \tau}$$

$$\begin{array}{lll} A \vdash x^\tau : \tau \text{ if } x^\tau \in A & A \vdash \ulcorner n \urcorner : o \text{ for all } n & A \vdash \mathsf{Y}^\tau : (\tau \to \tau) \to \tau \\ A \vdash \mathsf{add1} : o \to o & A \vdash \mathsf{sub1} : o \to o & A \vdash \mathsf{if0} : o \to o \to o \to o \\ A \vdash \mathsf{call/cdc} : ((o \to o) \to o) \to o & A \vdash \mathsf{error}_1 : o & A \vdash \mathsf{error}_2 : o \end{array}$$

FIGURE 1: SPCF: Syntax

procedure types $(\sigma \to \tau)$. In conformance with the usual convention, the type expression $\tau \to \sigma \to \nu$ abbreviates $\tau \to (\sigma \to \nu)$. The typing rules for SPCF are similar to those of the simply typed λ-calculus, but in SPCF every well-typed phrase M has a unique type τ. In the typing rules given in Figure 1, the only interesting clauses are the ones on the last line, which handle the additional constructs of SPCF. Λ^τ is the sub-language of SPCF that is built without constants.

Functional and Syntactic Extensions. When we write SPCF programs, we freely use functions and syntactic abbreviations that are easily definable in the language. Specifically, we will assume that the names $+$ and $-$ denote some addition and subtraction functions. When we use these names, the evaluation order of the arguments for these functions is unimportant. We will also use the following syntactic abbreviations:

$$\begin{array}{rcl} \Omega & \stackrel{df}{=} & \mathsf{Y}(\lambda x.x) \\ \textbf{let } x = M \textbf{ in } N & \stackrel{df}{=} & ((\lambda x\,.\,N)\,M) \\ \textbf{let}{*}\ x = M\ \ldots \textbf{ in } N & \stackrel{df}{=} & ((\lambda x\,.\,\textbf{let}{*}\ \ldots\ \textbf{in } N)\,M) \\ \textbf{let}{*}\ x = M \textbf{ in } N & \stackrel{df}{=} & ((\lambda x\,.\,N)\,M) \\ \textbf{letrec } L = M \textbf{ in } (L\ N) & \stackrel{df}{=} & \mathsf{Y}\,(\lambda L\,.\,M)\,N \end{array}$$

and some occasional syntactic extensions that are self-explanatory. ∎

The meaning of the PCF sub-language of SPCF is the usual one, *i.e.*, numerals and functional constants have their expected behavior, λ-abstractions are call-by-name procedures, and juxtaposition is function application. The meaning of the SPCF primitives not present in PCF is easy to describe informally; Section 3 contains a denotational, Section 4 an operational description.

The error constants, error_1 and error_2, generate special "error" values that are propagated according to the usual by-need evaluation order in call-by-name programs. If an SPCF procedure uses an argument and the by-need evaluation of the argument generates an error value e, the procedure returns the value e. On the other hand, if the procedure ignores an argument, the meaning of the argument—even if it generates an error—is irrelevant because by-need evaluation never evaluates the argument.

SPCF's call/cdc procedure of type $(o \to o) \to o$ is similar to Scheme's call/cc procedure. It applies its argument to the current continuation, but unlike full-fledged call/cc-style continuations, call/cdc-continuations can be used at most once.[1] Specifically, its type prevents call/cdc from returning the continuation or a procedure invoking the continuation.

To illustrate the use of error elements and call/cdc for observing the sequentiality of procedures, consider the following two definitions of a binary addition procedure in SPCF:

$$\begin{aligned} +_l &= \mathsf{Y}(\lambda f\,.\,(\lambda xy\,.\,\mathsf{if0}\ x\ y\ (\mathsf{add1}\ (f\ (\mathsf{sub1}\ x)\ y)))) \\ +_r &= \mathsf{Y}(\lambda f\,.\,(\lambda xy\,.\,\mathsf{if0}\ y\ x\ (\mathsf{add1}\ (f\ x\ (\mathsf{sub1}\ y))))) \end{aligned}$$

The first version $+_l$ recurs on the first argument (x); the second version recurs on the second argument (y). In contrast to PCF where the two procedures are observationally indistinguishable, SPCF provides several ways to distinguish these procedures. To begin with, the two procedures could be applied to error-generating arguments: $(f\ \mathsf{error}_1\ \mathsf{error}_2)$. When f is replaced by $+_l$ the resulting program will produce error_1, otherwise, when f is replaced by $+_r$, the result will be error_2. While these observations are accessible to the user of the program, the program itself cannot exploit them. A program can find out which addition procedure is used by applying the following procedure:

$$\mathit{Distinguish} = \lambda f.(\mathsf{call/cdc}(\lambda t.(f\ (t\ \ulcorner 0 \urcorner)\ (t\ \ulcorner 1 \urcorner)))).$$

On the input $+_l$, it yields $\ulcorner 0 \urcorner$; on the input $+_r$, it yields $\ulcorner 1 \urcorner$.

Because SPCF can determine which schedule a procedure uses to evaluate its arguments, the denotations of these procedures are *observably sequential* functions, a sub-domain of the continuous functions. In the next section, we describe the construction of a model for SPCF and discuss its important properties.

Constructs for non-local control: catch versus call/cdc. The original version of SPCF [9] used a family of catch procedures instead of call/cdc. The catch family includes a control operator catch^τ of type $\tau \to o$ for every type τ. The decision tree representing catch^τ depends only on the number k of inputs in type

$$\tau = \tau_1 \to \ldots \to \tau_k \to o\,.$$

The two constructs, catch and call/cdc, are interdefinable. Given call/cdc, it is trivial to define the catch^τ procedure for each type τ by the equations

$$\begin{aligned} \mathsf{catch}_0 &= \lambda x\,.\,x \\ \mathsf{catch}_k &= \lambda f\,.\,\mathsf{call/cdc}(\lambda t.(+\ \ulcorner k \urcorner\ (f\ (t\ \ulcorner 0 \urcorner)\ldots(t\ \ulcorner k-1 \urcorner)))) \end{aligned}$$

[1] Operationally speaking, the continuation can only be used to erase portions of the control stack in a *downward* fashion.

where the subscript k indicates the number of inputs in type τ.

Conversely, given the family catch procedures, the function call/cdc is defined by the equation follows:

$$\begin{array}{lll} \mathsf{call/cdc} & = & \lambda f.\mathbf{letrec}\ L = \lambda v.\mathbf{let}\ test_v = \lambda xy.(f(\lambda a\,.\,\mathsf{if0}\ (-\ a\ v)\ x\ y)) \\ & & \qquad\qquad\qquad\quad \mathbf{in\ let}\ root = (\mathsf{catch}\ test_v) \\ & & \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\mathsf{if0}\ root\ v \\ & & \qquad\qquad\qquad\qquad\qquad\quad (\mathsf{if0}\ (\mathsf{sub1}\ root)\ (L\ (\mathsf{add1}\ v))\ (-\ root\ \ulcorner 2 \urcorner))) \\ & & \qquad \mathbf{in}\ (L\ \ulcorner 0 \urcorner) \end{array}$$

The call/cdc procedure repeatedly explores its argument f until it finds the value $\ulcorner n \urcorner$ that f first passes to its argument (the continuation), which it returns as the answer. If f does not call its argument, call/cdc returns f's result. ∎

3 The Observably Sequential Type Frame

To define the semantics for SPCF we need

1. a collection *Dom* of *domains*, one for each type τ;
2. a meaning function $\mathcal{T} : \mathit{Types} \to \mathit{Dom}$ to assign meaning to type expressions;
3. a meaning function $\mathcal{C} : \mathit{Const}_\tau \to \mathcal{T}[\![\tau]\!]$ to assign meaning to constants of type τ in the domain $\mathcal{T}[\![\tau]\!]$; and
4. a meaning function $\mathcal{M} : \mathit{Terms}_\tau \to \mathit{Env} \to \mathcal{T}[\![\tau]\!]$ to assign meaning to terms of type τ in the domain for τ.

To describe the error values and control operators of SPCF, we must build domains with a topological structure that reflects the observably sequential behavior of computations. This section describes how to construct these domains, which are called *observably sequential domains* (or *OS*-domains). It also shows how to use these domains to build a denotational model for SPCF.

3.1 *OS*-Domains

A *Scott domain* $(D, \sqsubseteq)$ is the ideal completion of a *Scott basis* $(E, \leq)$. A *Scott basis* $(E, \leq)$ is a partial order closed under least upper bounds for bounded subsets. The *finite elements* of the domain are the principal ideals generated by the elements of the Scott basis E. Given Scott domains A and B, the set $A \to_c B$ of *continuous* functions forms a Scott domain under the pointwise ordering: $f \sqsubseteq g \Leftrightarrow \forall x \in A : f(x) \sqsubseteq_B g(x)$.

OS-domains are Scott domains that satisfy certain topological constraints and include a sufficient number of error (generating) elements. To specify the topological constraints, we need to identify a restricted set of finite elements called the *prime basis*. In fact, the simplest way to describe an *OS*-domain is to start with a prime basis and construct the domain by a process similar to ideal completion.

A prime basis is a partial order partitioned into *conflict* sets of pairwise unbounded elements. To account for the behavior of *errors*, we need to ensure that the prime basis is *error-rich*. To formalize the definition of a prime basis, we need to define the *precedes* relation $\prec$ and the *unique predecessor property.*

Definition 3.1. (*Unique Predecessor Property*) Recall that in a partial order $(P, \leq)$, an element p *immediately precedes* an element q (written $p \prec q$) iff (i) $p < q$ and (ii) $p \leq r < q$ implies that $r = p$. An element p such that $p \prec q$ is called a *predecessor* of q. The partial order $(P, \leq)$ has the *unique predecessor property* if every element $p \in P$ has at most one predecessor. ∎

Given the preceding definition, we can succinctly define the notion of a prime-basis as follows.

Definition 3.2. ((*Error-rich*) *Prime Basis*) A triple $(P, \leq, C)$ is a *prime basis* iff

[PO] $(P, \leq)$ is a countable partial order;

[UP] $(P, \leq)$ satisfies the unique predecessor property;

[C] the set $C = \{Q_1, Q_2, \ldots\}$ of *conflict* sets (or C-sets) is a partitioning of P such that $p, q \in C_i$ implies that p and q have the same set of predecessors;

[F] only a finite number of elements in P approximate each element of P.

A prime basis $(P, \leq, C)$ is *error-rich* iff

[E] each C-set $Q \in C$ has two designated *maximal* elements error_1^Q, error_2^Q.

We refer to C-sets by capital letters such as Q, R and S; we refer to elements of C-sets by the letters q, r, and s. We will augment these names by prime $(')$ superscripts and integer subscripts. The designated error elements play an essential role in the definition of *observably sequential functions* given in Section 3.2. ∎

An observably sequential domain (*OS*-domain) is the "prime-closure" of an error-rich prime basis.

Definition 3.3. (*Prime-Domain, OS-domain*) A *prime-domain* is a partial order $(D, \sqsubseteq)$ generated from a prime basis $(P, \leq, C)$ where

- the elements of D are *downward-closed* subsets of P:

$$I \in D \Leftrightarrow I \subseteq P \text{ and } \forall x \in I, y \in P : \text{ if } y \leq x \text{ then } y \in I\,;$$

- the elements of D do not contain conflicting primes:

$$\forall I \in D,\, Q \in C\ x,\, y \in I : \{x, y\} \not\subseteq Q\,;$$

  and

- the relation $\sqsubseteq$ is the subset ordering $\subseteq$ on D.

The domain generated by the prime basis $(P, \leq, C)$ is denoted $\mathbb{D}(P, \leq, C)$. If the prime basis $(P, \leq, C)$ is error-rich, then the prime-domain $\mathbb{D}(P, \leq, C)$ is an OS-domain. We refer to domain elements in an OS-domain by the letters d and e (with primes and indices). ∎

It is easy to show that every prime-domain $(D, \sqsubseteq)$ is a Scott domain; the finite sets of primes in D are the finite elements. As is customary in denotational semantics, we will identify all Scott domains that are order-isomorphic, eliminating the distinction between a prime basis B and the corresponding finite elements in the domain $\mathbb{D}(B)$.

Given an arbitrary Scott domain S, it is easy to determine whether S is a prime-domain. First, we identify the set P of *prime* elements of S, the (finite) elements other than $\perp$ that are not least upper bounds of sets of other elements. Second, we confirm that the prime elements form a prime basis $(P, \leq, C)$ where C is the partitioning of P determined by the relation

$$p \sim q \quad \Leftrightarrow \quad p = q \text{ or } \{p, q\} \text{ is unbounded and } p \text{ and } q \text{ have the same predecessors in } S\,.$$

Finally, we must verify that the "prime-closure" of $(P, \leq, C)$ is order-isomorphic to the original domain.

If S is a prime-domain with designated error elements, then it is an OS-domain iff the prime basis is error-rich, *i.e.*, exactly two maximal elements in each C-set are designated as error elements (property [**E**]).

Most domains encountered in the literature are either prime domains or domains that are *weakly generated* by their prime elements. In the latter case, every element of the domain is the least upper bound of the set of primes that approximate it, but the primes do not necessarily form a prime basis.

Consider the following simple examples.

1. The flat domain $\mathbb{T}_\perp$ of truth values $\{\mathsf{tt}, \mathsf{ff}\}$ is a prime-domain; the truth values $\{\mathsf{tt}, \mathsf{ff}\}$ are the prime elements.

2. The flat domain $\mathbb{N}_\perp$ of natural numbers is a prime-domain; every number is a prime element.

3. The continuous function space $\mathbb{T}_\perp \to_c \mathbb{T}_\perp$ is not a prime-domain, but it is weakly generated by its prime elements, which are the "one-step" functions:

$$(d \Rightarrow e)(x) \stackrel{df}{=} \begin{cases} e & \text{if } d \sqsubseteq x \\ \perp & \text{otherwise} \end{cases}$$

   where $d \in \mathbb{T}_\perp$, $e \in \mathbb{T}$. $\mathbb{T}_\perp \to \mathbb{T}_\perp$ is not a prime-domain, because it violates property [**C**].

4. The continuous function space $\mathbb{N}_\perp \to_c \mathbb{N}_\perp$ is not a prime-domain, but it is weakly generated by its prime elements, which are the "one-step" functions:

$$(d \Rightarrow e)(x) \stackrel{df}{=} \begin{cases} e & \text{if } d \sqsubseteq x \\ \perp & \text{otherwise} \end{cases}$$

where $d \in \mathbb{N}_\perp$, $e \in \mathbb{N}$. $\mathbb{N}_\perp \to \mathbb{N}_\perp$ is not a prime-domain, because it violates both property [**F**] and property [**C**]. The prime element $(\perp \Rightarrow 1)$ has infinitely many prime elements $(n \Rightarrow 1)$ $(n \in \mathbb{N})$ below it.

In the sequel, we will use "one-step" functions as parts of prime functions over *OS*-domains. As we saw in the examples above, the prime elements of non-trivial continuous function spaces do not form a prime basis, because they fail property [**C**]. They also fail property [**F**] if the input space is infinite.

A simple example of an observably sequential domain (a prime-domain with designated error elements) is $\mathbb{N}^E_\perp$, the flat domain of natural numbers plus two error elements, error_1 and error_2. It is generated by the prime basis $(\mathbb{N}^E, =, C)$ where

$$\begin{aligned} \mathbb{N}^E &= \mathbb{N} \cup \{\mathsf{error}_1, \mathsf{error}_2\} \\ C &= \{\{n\} \mid n \in \mathbb{N}^E\} \end{aligned}$$

We will use this domain later to assign meaning to the SPCF type o.

A more interesting example of observably sequential domain is the domain of functions generated by error-rich prime basis is $(P_{os}, \leq_{os}, C_{os})$ where P_{os} consists of three disjoint subsets of continuous functions from $\mathbb{N}^E_\perp$ to $\mathbb{N}^E_\perp$:

Constant primes generate outputs independent of their inputs. Since a function of this form ignores its inputs, it never returns an error value unless its constant output is an error value. For every $e \in \mathbb{N}^E$, there exists one constant prime:

$$g_e = (\perp \Rightarrow e).$$

In SPCF, the procedure $\lambda x.\ulcorner 5 \urcorner$ denotes the prime g_5.

Strict one-step primes output designated elements if their inputs contain enough information. If the input is an error value error_i, a strict one-step prime returns error_i. For every pair $d \in \mathbb{N}, e \in \mathbb{N}^E$, there is a strict, one-step prime:

$$f_{d,e} = (d \Rightarrow e) \sqcup (\mathsf{error}_1 \Rightarrow \mathsf{error}_1) \sqcup (\mathsf{error}_2 \Rightarrow \mathsf{error}_2).$$

In SPCF, the procedure $\lambda x.(\mathsf{if0}\ (=\ x\ \ulcorner d \urcorner)\ \ulcorner e \urcorner\ \Omega)$ denotes the strict one-step prime $f_{d,e}$ for $d, e \in \mathbb{N}$.

The diverging prime s inspects its inputs and then diverges. If the input is an error value e, the function s returns e:

$$s = (\perp \Rightarrow \perp) \sqcup (\mathsf{error}_1 \Rightarrow \mathsf{error}_1) \sqcup (\mathsf{error}_2 \Rightarrow \mathsf{error}_2).$$

In SPCF, the procedure $\lambda x.\mathsf{if0}\ x\ \Omega\ \Omega$ denotes the diverging prime function.

In summary, $P_{os} = \{g_e, f_{d,e}, s \mid d \in \mathbb{N}, e \in \mathbb{N}^E\}$. The approximation ordering on P_{os} is the usual pointwise approximation ordering on functions. The conflict partitioning for P_{os} is:

$$C = \{B, C_k \mid k \in \mathbb{N}\}$$

where

$$B \stackrel{df}{=} \{s, g_e \mid e \in \mathbb{N}_\perp^E\}$$
$$C_k \stackrel{df}{=} \{f_{k,e} \mid e \in \mathbb{N}_\perp^E\}$$

The designated error elements in B and C are g_{error_i} and f_{error_i}, respectively.

Technically, each element in the prime domain $\mathbb{D}(P_{os}, \leq_{os}, C_{os})$ is a set of functions rather than a function. However, we can identify each such set of functions with its least upper bound under the usual pointwise ordering on functions. Since every function in P_{os} is continuous, these bounding functions are also continuous.

A simple example is the set $\{s, f_{k,k} \mid k \in \mathbb{N}\} \in \mathbb{D}(P_{os}, \leq_{os}, C_{os})$, which has the identity function as its least upper bound. Similarly, the set $\{s, f_{k,k+1} \mid k \in \mathbb{N}\}$ corresponds to the successor function.

The domain $\mathbb{D}(P_{os}, \leq_{os}, C_{os})$ is a simple example of an *observably sequential* function space. To describe the topological structure of an arbitrary observably sequential function space, we need to introduce the concepts of "coverage", "direction", and "distance".

Definition 3.4. (*Coverage, Direction, Distance*) Let $(P, \leq, C)$ be a prime basis. A C-set Q *covers* a *prime element* $p \in P$ (written $p \prec Q$) iff there is a $q \in Q$ such that $p \prec q$ (implying that $p \prec r$ for all $r \in Q$). A C-set Q *covers* an *element* a in $\mathbb{D}(P, \leq, C)$ (written $a \prec Q$) iff there exists a prime $p \in a$ such that

$$(Q \cap a = \emptyset) \text{ and } (p \prec Q).$$

We say that Q *covers* a.

The *direction* relation is the transitive generalization of the *coverage* relation between primes and C-sets. More precisely, a prime element p has *direction* Q, where $Q \in C$ iff $\exists q \in Q : p \leq q$. We extend this notion to a relation on C-sets by saying that a C-set R is in the direction Q if there is a prime in R in the direction Q.

In conformance with the notation $p \leq Q$, we write $p \leq Q$ when $p \leq q \in Q$.

If p and q are primes in a prime-domain D such that $p \leq q$, the *distance from* p to q is the cardinality of the set of all the primes that approximate q, but not p. We extend this notion to C-sets by defining the distance between C-sets Q and R as the distance between any two primes $q \in Q$ and $r \in R$ such that $q \leq r$. ∎

A C-set Q covers a finite element a iff for any element $q \in Q$, $\{q\} \cup a$ is an element of the domain and $a \prec \{q\} \cup a$. Hence, the elements of a C-set Q covering a are mutually exclusive bits of information that can be incrementally added to the set of information a. In general, a finite element a may be covered by several different C-sets.

In the observably sequential domain generated by $(P_{os}, \leq_{os}, C_{os})$, the C-set B does not cover any prime elements. On the other hand, C_k covers the element s.

3.2 *OS*-Function Spaces

The *OS*-functions between two *OS*-domains A and B are a subset of continuous functions $A \to B$. Like their operational counterparts in SPCF, observably sequential functions propagate error values returned by *inspected* inputs. Since inputs are inspected only when they are "needed", we must formalize the concept of "need" to understand the semantics of error propagation. The notion of a sequentiality index [4, 17] captures the informal idea of "need" in a general context. The following definition adapts this notion to *OS*-domains.

Definition 3.5. (*Sequentiality Index*) Let $\mathbb{D}_1(P_1, \leq_1, C_1), \mathbb{D}_2(P_2, \leq_2, C_2)$ be *OS*-domains. For a continuous function $f : D_1 \rightarrow_c D_2$, a *sequentiality index* for finite input $a \in D_1$ and output C-set Q_2 covering $f(a)$ is a C-set R_1 covering a such that for all $x \sqsupset a$, $(f(x) \cap Q_2) \neq \emptyset$ implies $(x \cap R_1) \neq \emptyset$. ∎

The presence of the error elements in the domain force a function to indicate which C-set above a is explored first. If R_1 is a sequentiality index of f at a and Q_2, then f determines which element $r \in R_1$ approximates f's input $x \sqsupset a$ when it generates the output prime in Q_2. If f propagates errors, then it must generate the output prime $\mathsf{error}_i^{Q_2}$ if r is $\mathsf{error}_i^{R_1}$. Such a function implicitly has its evaluation strategy embedded in its graph, and hence it is called observably sequential function or *OS*-function for short.

Definition 3.6. (*Observably Sequential Function*) Let D_1 and D_2 be *OS*-domains, generated from the bases $(P_1, \leq_1, C_1)$ and $(P_2, \leq_2, C_2)$, respectively. A continuous function $f : D_1 \rightarrow_c D_2$ is an *OS*-function iff it is

sequential: for every pair (a, Q_2), where Q_2 covers $f(a)$, there is a sequentiality index R_1 if $q \in f(x)$ for some $q \in Q_2$ and $x \sqsupset a$; and

error sensitive: if R is the sequentiality index of f for the input a and the output C-set Q, $\mathsf{error}_1^Q \in (f(a \cup \mathsf{error}_1^R))$ and $\mathsf{error}_2^Q \in f(a \cup \mathsf{error}_2^R)$.

We use $D_1 \rightarrow_{os} D_2$ to denote the set of the *OS*-functions between two *OS*-domains D_1, D_2. ∎

An *OS*-function $f : D_1 \rightarrow_{os} D_2$ has at most one sequentiality index for a given finite input $a \in D_1$ and C-set Q_2 covering $f(a)$ [10]. This property is a direct consequence of the fact that f propagates errors. Therefore, for *OS*-functions, it makes sense to introduce the function si where $si_f(a, Q)$ is the sequentiality index of f for a and Q.

A simple example of an *OS*-function space is $\mathbb{D}(P_{os}, \leq_{os}, C_{os})$ defined in the previous subsection. It is isomorphic to the function space $\mathbb{N}_\perp^E \rightarrow_{os} \mathbb{N}_\perp^E$. We will prove that the *OS*-functions between two *OS*-domains form an *OS*-domain under the usual pointwise approximation ordering.

The following lemma reduces the observable sequentiality of an arbitrary function in $D_1 \rightarrow_c D_2$ to the observable sequentiality of its finite approximations:

Lemma 3.7 *A function $f : D_1 \rightarrow_c D_2$ is observably sequential iff every finite approximation $g \sqsubseteq f$ in $D_1 \rightarrow_c D_2$ is observably sequential.*

Proof. The lemma is an immediate consequence of the definitions of observable sequentiality and the approximation ordering on $D_1 \rightarrow_c D_2$. ∎

To prove that $D_1 \rightarrow_{os} D_2$ is an *OS*-domain, we need to identify a prime basis for the space, specifically, the set of primes and the conflict sets.

Definition 3.8. (*Prime Basis for* $\rightarrow_{os}$) Let $D_i(P_i, \leq_i, C_i$ be *OS*-domains, for $i = 1, 2$. The partial order $(P_{\rightarrow}, \leq_{\rightarrow})$ consists of special functions in $D_1 \rightarrow_{os} D_2$ that map finite elements of D_1 to prime elements of D_2. There are two disjoint subsets of primes:

1. **Output Primes:** an *OS*-function $f : D_1 \rightarrow_{os} D_2$ is an *output* prime function iff there exists a finite element $a \in D_1$ and a prime $p \in P_2$ such that f is a minimal function in $D_1 \rightarrow_{os} D_2$ satisfying the constraint $f(a) = p$ and $\forall a' \subset a\ f(a') \subset f(a)$. The function f has the C-set *label* $\langle a, Q_2, F'\rangle$ where $Q_2 \in C_2$ is the C-set containing p and $F' = \{g \in D_1 \rightarrow_{os} D_2 \,|\, g \subset f\}$.

2. **Schedule Primes:** an *OS*-function $f : D_1 \rightarrow_{os} D_2$ is a *schedule prime* iff there exists a finite element $a \in D_1$, a C-set $R_1 \in C_1$ where $a \prec R_1$, and a C-set $Q_2 \in C_2$ such that f is a minimal function with the property $f(a \sqcup \mathsf{error}_i^{R_1}) = \mathsf{error}_i^{Q_2}$ and $f(a) \subset \mathsf{error}_1^{Q_2}$. The function f has the C-set *label* $\langle a, Q_2, F'\rangle$ where $F' = \{g \in D_1 \rightarrow_{os} D_2 \,|\, g \subset f\}$.

$C_{\rightarrow}$ is the partitioning of $P_{\rightarrow}$ determined by the equivalence relation $\sim$ on $P_{\rightarrow}$ where

$$p \sim q \Leftrightarrow \text{ the } \mathsf{C}\text{-set } \textit{label} \text{ of } p = \text{ the } \mathsf{C}\text{-set } \textit{label} \text{ of } p.$$

The two designated error elements of a C-set with label $\langle a, Q, F'\rangle$ are $F' \cup \{(a, \mathsf{error}_1^Q)\}$ and $F' \cup \{(a, \mathsf{error}_2^Q)\}$. ∎

It is easy to show that $(P_{\rightarrow}, \leq, C_{\rightarrow})$ forms a prime basis.

Lemma 3.9 *$(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ is an error-rich prime basis.*

Proof Sketch. We prove that $(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ satisfies each of the properties of a prime basis. The verification of the conditions [**PO**], [**UP**], and [**C**] is straightforward. Also, each C-set has two maximal primes designated as error_1 and error_2. Finally, as to [[**F**]], let $\langle a, Q_2, F'\rangle$ be the C-set label of p. The function p needs to perform only a finite number of "actions" looking for each prime in a in the input and generating each prime approximating Q_2 in the output. Since each action corresponds to a prime function, the prime functions approximating p are equal to the number of actions, that is, number of primes in a plus the number of primes approximating Q_2. Hence, the property [**F**] holds for the prime functions. ∎

In the sample *OS*-function space $\mathbb{N}_\perp^E \rightarrow_{os} \mathbb{N}_\perp^E$ given in Section 3.1, the output primes are $\{g_e, f_{d,e} \,|\, d \in \mathbb{N}, e \in \mathbb{N}^E\}$ and the sole schedule prime is s. The output primes approximating a function determine what output the function produces for a given input. But this information does not completely describe the behavior of the function; it leaves open the possibility that a function could explore several different sequentiality indices simultaneously, permitting an output prime to be generated by either of two consistent pieces of information. To avoid this ambiguity, schedule primes uniquely determine the sequentiality index of a function. Specifically, the schedule prime determined by F', a, R_1, and Q_2 has the unique sequentiality index R_1 at a and Q_2; the set F' is the chain of approximations.

Theorem 3.10 *Let $\mathbb{D}(P_1, \leq_1, C_1)$ and $\mathbb{D}(P_2,\ \leq_2,\ C_2)$ be* OS*-domains. Then, the function space $\mathbb{D}(P_1, \leq_1, C_1) \rightarrow_{os} \mathbb{D}(P_2,\ \leq_2,\ C_2)$ is isomorphic to the domain $\mathbb{D}(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$. Each set of functions $d \in \mathbb{D}(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ is identified with its least upper bound in $\mathbb{D}(P_1, \leq_1, C_1) \rightarrow_c \mathbb{D}(P_2,\ \leq_2,\ C_2)$.*

Proof Sketch. The proof has two parts. First, we prove that any function generated from the prime basis $(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ is observably sequential. Second, we prove that any OS-function is the least upper bound of all its prime approximations in $P_{\rightarrow}$. These two facts imply that $(P_{\rightarrow}, \leq_{\rightarrow}, C_{\rightarrow})$ generates the OS-function domain $D_1 \rightarrow_{os} D_2$.

Claim 1: *The least upper bound in $D_1 \rightarrow_c D_2$ of a non-conflicting, downward-closed subset $S \subseteq P_{\rightarrow}$ is observably sequential.*

By Lemma 3.7, the proof of **Claim 1** reduces to proving that every bounded pair of finite OS-functions in $D_1 \rightarrow_c D_2$ has a least upper bound in $D_1 \rightarrow_c D_2$.

Let f and g be two OS-functions in $P_{\rightarrow}$ with least upper bound h in $D_1 \rightarrow_c D_2$. Let (a, Q_2) be a pair where Q_2 covers $h(a)$. If $h(x) \cap Q_2 \neq \emptyset$ and $x \supset a$, we need to show that h has a unique sequentiality index R_1 at (a, Q_2). We also need to show that $\mathsf{error}_i^{Q_2} \in h(a \sqcup \mathsf{error}_i^{R_1})$ for $i = 1, 2$.

Let $R_f = si_f(a, Q_2)$ and $R_g = si_g(a, Q_2)$. Either $R_f = R_g$ or only one of them exists. If $R_f \neq R_g$, then f and g can not be bounded above by any continuous function. If neither R_f nor R_g exists, then for all x, $h(x) \not\supset h(a)$.

Without loss of generality, assume that R_f is defined. Clearly, $R_f = si_h(a, Q_2)$ because $h(a \sqcup \mathsf{error}_1^{R_f}) \supseteq f(a \sqcup \mathsf{error}_i^{R_f})$. Since f is observably sequential, $\mathsf{error}_i^{Q_2} \in h(a \sqcup \mathsf{error}_1^{R_f})$. $\blacksquare_1$

Claim 2: *Any* OS*-function is the least upper bound of its prime approximations.*

Let the set $S_f = \{p \in P_{\rightarrow} \mid p \sqsubseteq f\}$. We will show that

$$\forall x \in D_1 : f(x) = \bigsqcup \{p(x) \mid p \in S_f\}.$$

Set $z = \bigsqcup S_f(x)$. It is clear that $z \subseteq f(x)$. To prove the inclusion in the opposite direction, we observe that if $z \subset f(x)$, there must be a prime element q in $f(x)$ that is absent in z. Without loss of generality, we assume that x is a minimal element that has q in $f(x)$. Therefore x is a finite element in D_1. We will define a prime function g in S_f such that $g(x) = q$, contradicting the fact that $q \in f(x) \setminus z$.

If x does not have errors, we must define an output prime g such that $g(x) = q$. We can construct such an output prime $g \in S_f$ as follows:

$$\begin{array}{rcll} g(y) & = & f(y) \sqcap g(x) & \text{if } y \subseteq x \\ g(y \sqcup \mathsf{error}_i^{R_1}) & = & g(y) \sqcup \mathsf{error}_i^{Q_2} & \text{if } y \subset x, R_1 \text{ covers } y, Q_2 \leq q, \\ & & & \text{ and } \mathsf{error}_i^{Q_2} \in f(y \sqcup \mathsf{error}_i^{R_1}) \end{array}$$

If x has an error_i^R, then $g(x)$ must be error_i^Q. In this case, we must define a schedule prime g with final step $si_g(a, Q) = R$ where $a = x \setminus \{\mathsf{error}_i^R\}$. We can construct such a scheme prime $g \in S_f$ as follows:

$$\begin{array}{rcll} g(a \sqcup \mathsf{error}_i^R) & = & \mathsf{error}_i^Q & \\ g(y) & = & f(y) \sqcap \mathsf{error}_1^Q & \text{if } y \subseteq a \\ g(y \sqcup \mathsf{error}_i^{R_1}) & = & g(y) \sqcup \mathsf{error}_i^{Q_2} & \text{if } y \subseteq a, R_1 \text{ covers } y, Q_2 \leq \mathsf{error}_1^Q, \\ & & & \text{ and } \mathsf{error}_i^{Q_2} \in f(y \sqcup \mathsf{error}_i^{R_1}). \end{array}$$

Hence, $z \supseteq f(x)$, implying that **Claim 2** and thus the entire theorem are true. $\blacksquare$

3.3 Semantics of SPCF

Using the framework of *OS*-domains, we define a fully abstract "environment" model [18] for SPCF. The model is called the *Observably Sequential Type Frame*; its formal definition is given in Figure 2. The model maps each type τ to an *OS*-domain $\mathcal{T}[\![\tau]\!]$. The ground type (o) is interpreted as the flat domain of natural numbers with error values ($\mathbb{N}_{\perp}^{E}$), which is clearly a prime domain. The function types are domains of observably sequential function between appropriate spaces. An *environment* ρ is an observably sequential function mapping the flat *OS*-domain $\mathcal{V}_{\perp}^{E} = \mathcal{V} \cup \mathbb{E}_{\perp}$ of SPCF variable names to the union *Dom* of all domains $\mathcal{T}[\![\tau]\!]$ that respects the types of variable names: for all $x^{\tau} \in \mathcal{V}$, $\rho(x^{\tau}) \in \mathcal{T}[\![\tau]\!]$. It is easy to prove that the set of environments forms an *OS*-domain $\mathcal{E}$ under the pointwise approximation ordering. The function $\mathcal{C}$ maps constants of type τ to denotations in $\mathcal{T}[\![\tau]\!]$. Finally, if M is of type τ, the meaning function $\mathcal{M}$ assigns M the denotation $\mathcal{M}[\![M]\!] \in \mathcal{E} \to \mathcal{T}[\![\tau]\!]$.

The notation used in the definition follows the standard conventions in the literature. In particular, for a closed term M, we simply write $\mathcal{M}[\![M]\!]$ to denote the meaning of the term M—instead of $\mathcal{M}[\![M]\!]\rho$—because the environment ρ is irrelevant. By the same token, we write $\mathcal{M}[\![M]\!]$ to denote the meaning of an SPCF program M, since it is closed term of type o.

Figure 2 asserts that for any environment $\rho \in \mathcal{E}$ and a term M of type τ in SPCF, the meaning $\mathcal{M}[\![M]\!]\rho \in \mathcal{T}[\![\tau]\!]$. To prove this fact, we rely on standard techniques from Meyer [18] and Curien [11, 12]. In either case, the proof is laborious but uninteresting; the only interesting cases rely on the fact that *OS*-functions are closed under composition and λ-abstraction, which can be interpreted as a combinator [11, 7]. Following Curien's path shows that the set of *OS*-domains and *OS*-functions form a cartesian closed category.

Given a meaning function for SPCF, we can formulate the observational congruence relation ($\simeq_{SPCF}$). This relation captures the *behavioral* equivalences among terms and provides a mathematical justification for many compiler optimizations. It equates two terms precisely when an observer cannot tell the difference between these terms in arbitrary program contexts.

Definition 3.11. (*Observational Equivalence*) Let M and N be two SPCF terms of type τ. Then, M and N are observationally equivalent if for all contexts C^{τ} (programs with holes in places of terms of type τ) such that $C[M]$ and $C[N]$ are programs,

$$\mathcal{M}[\![C[M]]\!] = \mathcal{M}[\![C[N]]\!].$$

We write $M \simeq_{SPCF} N$ if M and N are observationally equivalent. ∎

The important property of the *OS*-domains model of SPCF is that two terms map to the same denotation (for all environments) if and precisely if they are observationally equivalent.

Theorem 3.12 *Let M and N be two SPCF terms of type τ. Then,*

$$M \simeq_{SPCF} N \text{ if and only if } \mathcal{M}[\![M]\!]\rho = \mathcal{M}[\![N]\!]\rho \text{ for all } \rho.$$

The original proof of this theorem for a concrete model of SPCF based on trees is due to Cartwright and Felleisen [9]. For the *OS*-domains model, the theorem follows from Theorem 5.8 in Section 5.

Domains

$$\begin{array}{rcl} \mathcal{T}[\![o]\!] & = & \mathbb{N}_\perp^E \\ \mathcal{T}[\![\sigma \to \tau]\!] & = & [\mathcal{T}[\![\sigma]\!] \to_{os} \mathcal{T}[\![\tau]\!]] \\ \mathcal{V}_\perp^E & = & \mathcal{V} \cup \mathbb{E}_\perp \\ \mathcal{E} & = & \{f \in \mathcal{V}_\perp^E \to_{os} \bigcup_\tau \mathcal{T}[\![\tau]\!] \mid f(x^\tau) \in \mathcal{T}[\![\tau]\!]\} \end{array}$$

where

$$\begin{array}{rcl} \mathbb{N}_\perp^E & \stackrel{df}{=} & \mathbb{N} \cup \mathbb{E}_\perp \\ \mathbb{E}_\perp & \stackrel{df}{=} & \{\perp, \mathsf{error}_1, \mathsf{error}_2\} \end{array}$$

Semantic functions

$$\begin{array}{rcl} \mathcal{C} & : & \mathit{Constants}_\tau \to \mathcal{T}[\![\tau]\!] \\ \mathcal{C}[\![\ulcorner n \urcorner]\!] & = & n \\ \mathcal{C}[\![\mathsf{error}_i]\!] & = & \mathsf{error}_i \\ \mathcal{C}[\![\mathsf{add1}]\!] & = & \{(\perp, \perp), (\mathsf{error}_i, \mathsf{error}_i), (l, l+1) \mid l \in \mathbb{N}\} \\ \mathcal{C}[\![\mathsf{sub1}]\!] & = & \{(\perp, \perp), (\mathsf{error}_i, \mathsf{error}_i), (0, \perp), (l+1, l) \mid l \in \mathbb{N}\} \\ \mathcal{C}[\![\mathsf{if0}]\!] & = & \{(\perp, l, m, \perp), (\mathsf{error}_i, l, m, \mathsf{error}_i), (0, l, m, l), (k+1, l, m, m) \mid l, m, k \in \mathbb{N}\} \\ \mathcal{C}[\![\mathsf{Y}]\!] & = & \underline{\lambda} f^{(\sigma\to\tau)\to(\sigma\to\tau)}. \bigsqcup_{n=0}^{\infty} f^n(\perp^{(\sigma\to\tau)}) \\ \mathcal{C}[\![\mathsf{call/cdc}]\!](f) & = & \begin{cases} n & ((\perp,\perp), n) \in f, ((\mathsf{error}_i, \mathsf{error}_i), n) \in f, \text{ or } ((n, \mathsf{error}_i), \mathsf{error}_i) \in f \\ \mathsf{error}_i & ((\perp,\perp), \mathsf{error}_i) \in f \end{cases} \end{array}$$

$$\begin{array}{rcl} \mathcal{M} & : & \mathit{Terms}_\tau \to_{os} \mathcal{E} \to_{os} \mathcal{T}[\![\tau]\!] \\ \mathcal{M}[\![b]\!]\rho & = & \mathcal{C}[\![b]\!] \\ \mathcal{M}[\![x]\!]\rho & = & \rho[\![x]\!] \\ \mathcal{M}[\![\lambda x.M]\!]\rho & = & (\underline{\lambda} v. \mathcal{M}[\![M]\!]\rho[x/v]) \\ \mathcal{M}[\![(M\ N)]\!]\rho & = & \mathcal{M}[\![M]\!]\rho(\mathcal{M}[\![N]\!]\rho) \end{array}$$

FIGURE 2: The semantics of SPCF

SPCF is more expressive than PCF. While SPCF is a conservative extension of PCF in the sense that it preserves the meanings of all proper PCF programs, its control operators add expressive power to PCF. More precisely, PCF cannot *simulate* SPCF's new features via "functions" or "macros": There is no syntactically homomorphic translation from SPCF to PCF that maps programs and sub-expressions that are already in PCF syntax to themselves. By Felleisen's theorem on expressiveness [13:Thm 3.14], this claim reduces to showing that the theory of observational equivalence for SPCF is not a superset of the theory of observational equivalence for PCF.

Based on the explanation of the possibilities of observing sequentiality at the end of Section 2, it is easy to prove that some of PCF's observational equalities no longer hold in

SPCF. Consider the equation:

$$(\lambda f.(f +_l)) \simeq_{PCF} (\lambda f.(f +_r))$$

In SPCF, we can use either error values or call/cdc to distinguish these functions. Both of the contexts

$$C_1 = ([\ ]\ (\lambda + .(+\ \mathsf{error}_1\ \mathsf{error}_2)))$$

and

$$C_2 = ([\ ]\ \mathit{Distinguish})$$

distinguish these functions. And hence, $(\lambda f.(f +_l)) \not\simeq_{SPCF} (\lambda f.(f +_r))$ as predicated. ∎

4 An SPCF Calculus

The λ-calculus for SPCF is an extension of the calculus for PCF [22]. The second part of Figure 3 contains the axioms of the PCF calculus: (β), (Y), and the axioms for PCF constants in the left-hand column. The SPCF calculus adds the four new axioms in the right-hand column of the second part of the same figure. Two axioms capture the behavior of each of the two error values, one that of a return and one that of an escape from a call/cdc-application.

Evaluation Contexts:

$$\begin{array}{rcl} E & ::= & [\ ] \mid (c^1\ E) \mid (c^3\ E\ M\ M) \mid (\mathsf{call/cdc}\ (\lambda x.E)) \\ c^1 & ::= & \mathsf{add1} \mid \mathsf{sub1} \mid \mathsf{call/cdc} \\ c^3 & ::= & \mathsf{if0} \end{array}$$

Axioms:

$$\begin{array}{rcll} (\lambda_n x.M)\ M' & = & M[x/M'] & (\beta) \\ \mathsf{Y}\ M & = & M\ (\mathsf{Y}\ M) & (\mathsf{Y}) \end{array}$$

$$\begin{array}{rcll} (\mathsf{add1}\ \ulcorner n \urcorner) & = & \ulcorner n+1 \urcorner & (\mathsf{add1}) \\ (\mathsf{sub1}\ \ulcorner n+1 \urcorner) & = & \ulcorner n \urcorner & (\mathsf{sub1}) \\ (\mathsf{if0}\ \ulcorner 0 \urcorner\ M\ N) & = & M & (\mathsf{if0}_0) \\ (\mathsf{if0}\ \ulcorner n+1 \urcorner\ M\ N) & = & N & (\mathsf{if0}_1) \end{array} \qquad \begin{array}{rcll} E[\mathsf{error}_1] & = & \mathsf{error}_1 & (\mathsf{error}_1) \\ E[\mathsf{error}_2] & = & \mathsf{error}_2 & (\mathsf{error}_2) \\ \mathsf{call/cdc}(\lambda f.\ulcorner n \urcorner) & = & \ulcorner n \urcorner & (\mathsf{return}) \\ \mathsf{call/cdc}(\lambda f.E[f\ \ulcorner n \urcorner]) & = & \ulcorner n \urcorner & (\mathsf{catch}) \\ & & \multicolumn{2}{l}{\text{if } f \text{ is not captured by } E} \end{array}$$

Semantics: $\mathit{eval}_{SPCF}(M) = \ulcorner m \urcorner$ if and only if $M = \ulcorner m \urcorner$

FIGURE 3: SPCF: Calculus and Semantics

The formalization of the additional axioms relies on the concept of an *evaluation context* [14, 15]. An evaluation context is a phrase with a hole at a position of ground type:

see the first part of Figure 3. Filling the hole with a phrase yields a complete phrase. The process of filling a hole may capture free variables that are bound as downward continuation variables in nested call/cdc-application.

The crucial property of the notion of evaluation context is that every phrase of ground type, is either a numeral, a variable, or has a unique decomposition into an evaluation context and a redex. The hole of the evaluation context specifies the position of the unique sub-phrase that *must* be evaluated in order to be able to evaluate the complete phrase. A *redex* R_E, relative to an evaluation context E, is either one of the following applications: (add1 $\ulcorner n \urcorner$), (sub1 $\ulcorner n \urcorner$), (if0 $\ulcorner n \urcorner$ M N), or (call/cdc $(\lambda x. \ulcorner n \urcorner)$); or it is an error value; or it is an application of an E-bound variable (f), a downward continuation, to a numeral: (f $\ulcorner n \urcorner$).

Lemma 4.1 (Unique Decomposition) *If M is an expression of ground type, then it is either a numeral, a variable, or there exists a* unique *evaluation context E and a redex R_E such that $M \equiv E[R_E]$.*

Proof Sketch. The proof is a simple induction on the structure of terms. ∎

If the hole of an evaluation context contains an error, the evaluation context is useless: it will never affect the evaluation of the program. If the hole contains the application of a downward continuation to a numeral ($\ulcorner n \urcorner$), the evaluation continues by eliminating the evaluation context and by returning $\ulcorner n \urcorner$ as the result of the (binding) call/cdc-application.

Given the SPCF calculus, the operational semantics of SPCF is specifed in the conventional fashion. A program, *i.e.*, a closed SPCF phrase of type o, evaluates to a numeral, if the SPCF calculus proves the equivalence between the two. To prove that this definition determines an evaluation *function*, we need to establish a Church-Rosser Theorem.

Theorem 4.2 (Consistency) *$eval_{SPCF}$ is a function.*

Proof Sketch. Let $\rightarrow_s$ denote the reduction system, based on reading the SPCF axioms from left to right. Then, it is easy to see that the theorem is implied by a Church-Rosser theorem for the reduction system. We already know that the PCF calculus is Church-Rosser. Based on the Unique Decomposition Lemma, we can easily prove that the reductions $\mathsf{error}_1, \mathsf{error}_2$ and catch, return satisfy the diamond property. Finally, the three reduction relations commute with each other, which by the Hindley-Rosen Lemma [1:3.3.5] proves that the complete system ($\rightarrow_s$) is Church-Rosser. (Also compare the proof of the Church-Rosser Theorem for the λ_v-C-calculus [14].) ∎

Given two distinct functions for determining a result for a program, it is natural to ask whether the functions are the same. The following theorem shows that the operational and the denotational semantics are indeed in complete agreement.

Theorem 4.3 (Adequacy) *For all SPCF programs M, $eval_{SPCF}(M) = \ulcorner m \urcorner$ iff $[\![M]\!] = m$.*

Proof Sketch. Curien [12] proved an adequacy theorem for SPCF with catch and for a *concrete* denotational model. As mentioned above, his concrete domains are isomorphic to our abstract domains. Hence, our denotational meaning functions agree. Furthermore, it is easy to show that Curien's calculus proves the same equations between programs that our calculus proves. Thus, our operational evaluators agree, too. ∎

Beyond its rôle in defining an operational semantics, a calculus is also a simple logic for reasoning about observational congruences. By the Adequacy Theorem, the axioms are clearly sound with respect to observational equivalence. The more interesting question, however, concerns the extent to which these axioms are *complete*. Since SPCF can clearly define all partial recursive functions on the natural numbers, the calculus cannot be an axiomatization for the entire theory. But, given the related results for λ-calculus-based languages, we can ask whether it is possible to axiomatize SPCF's observational congruence relation for certain sub-languages of SPCF.

The completeness question for the pure typed λ-calculus was first addressed by H. Friedman [16]. He proved that equality for the language Λ^τ in the full type hierachy is completely characterized by the $\boldsymbol{\lambda}\beta\eta$ calculus where (η) is the equational characterization of extensionality for Λ^τ:

$$(\lambda x.(M\ x)) \ = \ x \text{ if } x \text{ is not free in M} \qquad (\eta)$$

Plotkin [21] and Statman [23, 30, 31] later strengthened Friedman's theorem by proving that $\boldsymbol{\lambda}\beta\eta$ is also complete for the continuous type frame. Statman's 1-Section Theorem also provides a general proof method for determining whether $\boldsymbol{\lambda}\beta\eta$ is a complete axiomatization of equality for arbitrary extensional models of the simply typed λ-calculus. It follows from this latter theorem that we can completely axiomatize observational equality for the sub-language Λ^τ of SPCF.

Theorem 4.4 (Completeness for Λ^τ) *If $M, N \in \Lambda^\tau$, then*

$$\beta\eta \vdash M = N \ \textit{iff}\ [\![M]\!] = [\![N]\!] \ \textit{iff}\ M \simeq_{SPCF} N.$$

Proof Sketch. To prove the completeness of the $\lambda\beta\eta$ calculus for an extensional model according to Statman's 1-Section Theorem [30, 31, 23], we need to prove that there is a faithful embedding of the free algebra of binary trees over a single constant into the first-order portion of the model.

Recall the definition of the free binary tree algebra. It contains all finite terms over the signature $\{C^0, F^2\}$, i.e., it is the language generated by the production

$$T \to C \mid (F\ T\ T).$$

A faithful embedding of this algebra into a model requires an extension of the meaning function for constants, $\mathcal{C}$, that maps C to a natural number, $c \in \mathbb{N}$, and F to a pairing function,

$$p : \mathbb{N}_\perp^E \to_{os} \mathbb{N}_\perp^E \to_{os} \mathbb{N}_\perp^E,$$

such that $t_1 = t_2$ if and only if $\mathcal{M}[\![t_1]\!] = \mathcal{M}[\![t_2]\!]$.

For our purposes, we can take $c = 0$ and

$$p = \begin{cases} 2^x \cdot 3^y & \text{if } x, y \in \mathbb{N} \\ x & \text{if } x \in \mathbb{E}_\perp \\ y & \text{if } x \in \mathbb{N}, y \in \mathbb{E}_\perp \end{cases}$$

It is straightforward to show that an extension of $\mathcal{C}$ for C^0, F^2 based on c and p is faithful. ∎

In an upcoming paper, Riecke and Subrahmanyam [24] prove an extension of Statman's 1-Section Theorem to mixtures of Λ^τ and first-order algebras. Based on their central theorem [24:Thm. 11], it is possible to extend the above theorem to Λ^τ plus **add1** and the numerals. As to further extensions with first-order constants, their results show that the addition of sub1[2] or if0 based on their algebraic axioms does *not* yield a complete calculus.

A more interesting question with respect to SPCF's constants is whether the operational axioms for call/cdc completely capture its observable behavior. Unfortunately, this is not the case. Consider the following equation:[3]

$$(\lambda x^{o\to o}.(\mathsf{call/cdc}\ (\lambda y^{o\to o}.(x\ (x\ (y\ z)))))) \;=\; (\lambda x^{o\to o}.(\mathsf{call/cdc}\ (\lambda y^{o\to o}.(x\ (y\ z))))) \quad (\dagger)$$

It is easy to prove by a simple case analysis that the model validates this equation. If either of the two procedures is applied to a constant procedure f, the result is that of f. Otherwise, if they are applied to a strict procedure g, then they will both attempt to evaluate the sub-expression $(y\ z)$, which will cause them to return the value of z. Thus, the two procedures denote the same functions, and the equation holds in our model. The SPCF calculus, on the other hand, cannot prove the equation (†) because the two terms are in normal form and the underlying reduction system $(\to_s)$ is CR.

The counter-example reveals that the calculus cannot reason about the strictness behavior of variables. We conjecture that only an extension of the calculus to a sequent-style theory for case-based reasoning can completely axiomatize the equation that govern call/cdc's behavior.

5 The Computational Power of SPCF

Given a model $\mathcal{M}$ for a language L, it is natural to ask whether the language L can express all the *computable* elements in the model. More precisely, the question is whether every recursively enumerable element $d \in \mathcal{T}[\![\tau]\!]$ is definable by a term M of type τ in $\mathcal{L}$, *i.e.*, $\mathcal{M}[\![M]\!] = d$. When this property holds, we say that L is *computationally complete* for $\mathcal{M}$. Plotkin [22:Section 5][4] proved that PCF is not computationally complete for the familiar continuous function model. The continuous function model contains "parallel deterministic" functions that PCF *cannot* express. Two well-known independent examples are the parallel conditional function (pif0) and the parallel *exists* function ($\exists$). The pif0 function is similar to if0, but it returns the least upper bound of the two alternatives if the test argument diverges. The function $\exists$ tests whether its argument, a function from $\mathbb{N}_\perp$ to itself, produces tt (0 in our simplified version of PCF) for any input or whether it always returns ff.

[2]The problem with sub1 is due to the specification that PCF's sub1 diverges when applied to $\ulcorner 0 \urcorner$. Setting $(\mathsf{sub1}\ulcorner 0\urcorner) = \mathsf{error}_1$ also does not solve the problem because the presence of error values in the ground domain violates another antecedent of the Riecke-Subrahmanyam meta-theorem [Riecke: private communication, 15 October 1992].

[3]The idea for this equation is due to John Gateley.

[4]Also compare Sazonov's paper [25] on the expressibility of PCF. He proved that PCF can only express the sequential functions of its continuous function model, and conjectured that adding parallel-or and parallel-exists would make PCF a computationally complete language. His discovery of parallel-or and parallel-exists was independent of Plotkin's, but both operators were discussed in early manuscripts of D. Scott [Plotkin, Sazonov, & Scott: private communication, July 1992].

In contrast to PCF and PCF+**pif0**, SPCF does not need any extensions to be computationally complete. For any computable element in its natural model, there is a term that denotes this element. Given this fact, it is easy to show that the natural model of SPCF is fully abstract.

This section sketches a proof of the computational completeness of SPCF. The first subsection defines a generalization of Turing computability suitable for *OS*-domains. The second subsection applies this definition to SPCF and presents an algorithm for constructing an SPCF term that denotes a given domain element definable by a partial recursive function. This algorithm shows that SPCF is computationally complete for its *OS*-domains model.

5.1 Computability in Observably Sequential Domains

Computability over Scott domains is a generalization of the familiar notion of Turing computability over natural numbers. A computation over a Scott domain produces a recursively enumerable set of finite approximations whose least upper bound is the "answer" produced by the computation. In a computation over a Scott domain D, every finite element is represented by a distinct natural number. The specific choice of representation is determined by the *presentation* of the domain [27]; the presentation assigns a distinct integer code to each finite element and specifies the ordering relationship between any two finite elements.

5.1.1 Domain Presentations

When a Scott domain is an *OS*-domain, every finite approximation is the least upper bound of its prime approximations. Consequently, a *presentation* only needs to assign integer codes to the prime elements of an *OS*-domain. Given such a presentation, it is easy to generate unique codes for all finite elements using a standard pairing function.

We formalize the concept of a presentation of an *OS*-domain as follows:

Definition 5.1. (OS-*domain Presentation*) A *presentation* of an *OS*-domain D with prime-basis $(P, \leq, C)$ is a quadruple of functions $\langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{err} \rangle$:

- a one-to-one function $\mathbf{E} : P \to \mathbb{N}$, such that $p \leq_P q \Rightarrow \mathbf{E}(p) \leq_{\mathbb{N}} \mathbf{E}(q)$ that maps the elements of P into their codes in the index set in $\mathbb{N}$. By convention, we extend $\mathbf{E}$ to $\perp$ by defining $\mathbf{E}(\perp) \stackrel{df}{=} 0$. Let $\mathcal{I}$ be the range of the function $\mathbf{E}$.

- a function $\mathbf{c} : \mathcal{I} \to \mathbb{N}$ such that $\forall i, j > 0 : \mathbf{c}(i) = \mathbf{c}(j)$ iff $p_i, p_j \in Q$ for $Q \in C$; by convention, we define $\mathbf{c}(0) \stackrel{df}{=} 0$. Let the range of the function $\mathbf{c}$ be $\mathcal{S}$.

- a predecessor function $\mathbf{pr} : \mathcal{I} \to \mathcal{I}$ such that $\mathbf{pr}(i) = j$ iff $p_j \prec p_i$. If there is no predecessor for an element p_i, we define $\mathbf{pr}(i) \stackrel{df}{=} 0$. Similarly, we define $\mathbf{pr}(0) \stackrel{df}{=} 0$.

- a function $\mathbf{err} : \mathcal{I} \to \{0, 1, 2\}$ such that

$$\mathbf{err}(k) = \begin{cases} i & p_k = \mathsf{error}_i^Q \\ 0 & \text{otherwise}\,. \end{cases}$$

The presentation is *effective* iff the functions **c**, **pr**, **err** are partial recursive functions mapping $\mathbb{N}$ into $\mathbb{N}$. The function **E** is called a *prime enumerator* for D. ∎

The prime enumerator **E** maps each prime element to a unique index in $\mathbb{N}$. Hence, **E** determines an enumeration of prime elements $p_{i_1}, p_{i_2}, \ldots$ where $\mathbf{E}(p_{i_j}) = i_j$. For the sake of convenience, we require that the indexing determined by **E** be topologically sorted, which makes 0 the end-of-list marker.

The remaining three components of a presentation—**c**, **pr**, and **err**—specify how prime elements are partitioned in C-sets, what prime element precedes each C-set, and which prime elements are errors. The function **c** implicitly assigns a unique code number in $\mathcal{S} \subseteq \mathbb{N}$ to each C-set in the prime-basis. We refer to the C-set with index m as C_m. As usual, we remove the ambiguity by superscripting the C-sets with the domain whenever needed. This indexing scheme for C-sets determines the following partial order on $\mathcal{S}$, which we will use later in this section.

Definition 5.2. ($\leq'$) Let $m, m' \in \mathbb{N}$ be in the range of the function **c**. Then $m' \leq' m$ iff for all $q \in C_m$ there exists $p \in C_{m'}$ such that $p \leq q$. ∎

5.1.2 Encoding Finite Elements

Since finite elements are least upper bounds of finite sets of primes, it is easy to generate codes for the finite elements of $\mathbb{D}$ given **E**. The index of a finite element d is the integer code representing the finite set S_d of prime indices of primes below d. To make this encoding unique, we represent S_d as a sorted list of prime indices terminated by the *end-of-list* marker 0 and use a standard bijective pairing function to encode this list as an integer.

Definition 5.3. (*Index of finite element*) In the OS-domain with prime enumerator **E**, the finite element $a = \sqcup\{p_{k_1}, p_{k_2}, \ldots, p_{k_n}\}$ where $k_1 > k_2 > \ldots > k_n$ has the *index*

$$\langle k_1, \langle k_2, \ldots, \langle k_n, 0\rangle \ldots\rangle\rangle$$

where $\langle\cdot,\cdot\rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is the standard pairing function defined by

$$\langle i, j\rangle = \frac{(i+j) * (i+j+1)}{2} + i\,.$$

We denote the finite element with index I by a_I. We superscript the element by the domain it belongs to, if there is ambiguity.

Note: Each prime element has two indices: one as a prime element and another as a finite element. ∎

The encoding described is unique for every n-tuple of positive numbers $\langle k_1, k_2, \ldots, k_n\rangle$. Since the pairing function, the projection functions, and the sorting function over the encoded tuples are recursive, they can be expressed in SPCF as $\langle\cdot,\cdot\rangle$, **1st**, **2nd**, and **sort** respectively.

5.1.3 Computable Elements

We have now finally developed all of the machinery required to define the *computable* elements of an OS-domain. Informally, an element d of an effectively presented OS-domain D is computable iff there exists a partial recursive function that "represents" the element. The following definition makes this notion precise.

Definition 5.4. (*Computable Element (OS Domains)*) Let D be an OS-domain with the effective presentation $\langle \mathbf{E}, \mathbf{c}, \mathbf{pr}, \mathbf{err} \rangle$ where $\mathcal{I}$ denotes the range of $\mathbf{E}$ and $\mathcal{S}$ denotes the range of $\mathbf{c}$. An element d of an OS-domain D is *computable* iff the function $f_d : \mathcal{S} \multimap \mathcal{I}$ defined by

$$f_d(m) = \begin{cases} k & \text{if } \mathbf{c}(k) = m \text{ and } p_k \sqsubseteq d \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a partial recursive function over $\mathbb{N}$. We call the function f_d the *characteristic function* of d. ∎

5.1.4 A Comparison with Classical Domain Theory

The standard definition of computability for Scott domains is similar to our definition of computability for OS-domains. The only difference is that our definition exploits the additional topological structure of OS-domains. It is easy to show that an element d of an OS-domain D with effective presentation $\mathbf{E}$ is computable iff it is computable in the isomorphic Scott domain with the corresponding effective presentation.

Definition 5.5. (*Scott Presentation*) A *presentation* δ_s of a Scott domain D with basis $(B, \leq)$ is a triple $\langle \mathbf{E}_s, \mathbf{lub}, \mathbf{con} \rangle$ consisting of:

- a one-to-one onto function $\mathbf{E}_s : B \to \mathcal{I} \subseteq \mathbb{N}$, such that $\mathbf{E}_s(\bot) = 0$; $\mathbf{E}_s$ maps the elements of B into their codes in the index set $\mathcal{I} \subseteq \mathbb{N}$.
- a ternary predicate $\mathbf{lub} \subseteq \mathcal{I} \times \mathcal{I} \times \mathcal{I}$ such that
$$\mathbf{lub}(x, y, z) \Leftrightarrow b_x \sqcup b_y = b_z$$
where $b_x = E^{-1}(x), b_y = E^{-1}(y)$, and $b_z = E^{-1}(z)$;
- a binary predicate $\mathbf{con} \subseteq \mathcal{I} \times \mathcal{I}$ such that
$$\mathbf{con}(x, y) \Leftrightarrow \exists z \in B :\ x \sqsubseteq z \text{ and } y \sqsubseteq z\,.$$

The function $\mathbf{E}_s$ is called the *enumerator* for D. The presentation δ_s is *effective* iff the relations $\mathbf{lub}$ and $\mathbf{con}$ are recursive. ∎

Given an effective presentation δ for an OS-domain D, it is easy to generate the effective presentation δ_s for D viewed as a Scott domain.

We will show that d is a computable element an OS-domain D with effective presentation δ iff d is a computable element in the isomorphic Scott domain with effective presentation δ_s.

Definition 5.6. (*Computable Element (Scott Domains)*) Let D be a Scott domain with the enumerator $\mathbf{E}_s$, and let $\mathcal{I} = range(\mathbf{E}_s)$. An element d of the domain D is computable iff there is a partial recursive predicate g_d over $\mathcal{I}$ such that

$$g_d(i) = \begin{cases} \mathsf{tt} & \text{iff } a_i \sqsubseteq d \\ \text{undefined} & \text{otherwise} \end{cases}$$

We call g_d the (*Scott*) *characteristic function* for d. ∎

In an OS-domain D, the two definitions of computability coincide.

Lemma 5.7 *Let D be an* OS*-domain with effective presentation δ and let δ_s be the effective Scott for the isomorphic Scott domain D_s. An element $d_s \in D_s$ has a* Scott-characteristic function g_d *iff the corresponding element in $d \in D$ has a characteristic function f_d.*

Proof Sketch. ($\Leftarrow$) Given f_d we can define the corresponding Scott-characteristic function g_d as follows. Since

$$g_d(I) = \mathsf{tt} \Leftrightarrow g_d(k) = \mathsf{tt}\ \forall k : a_k \sqsubseteq a_I \text{ and } a_k \text{ is prime}$$

it suffices to define g_d on k such that a_k is prime. The function g_d is defined on *prime elements* a_k by the rule

$$g_d(k) = \mathsf{tt} \quad \Leftrightarrow \quad \forall j : p_j \sqsubseteq a_k : [f_d(\mathbf{c}(j)) = j] .$$

where the index k is the encoding of the tuple $\langle k_1, \ldots, k_n \rangle$. The set of prime indices $\{j \mid p_j \sqsubseteq a_k\}$ is simply $\{j_1, j_2, \ldots, j_n\}$ where $\langle j_1, \langle j_2, \ldots, \langle j_n, 0\rangle \ldots \rangle\rangle = k$. Hence, $g_d(k)$ is computable for k such that a_k is prime.

Conversely, given g_d, we can define f_d as follows. Given any prime index $k_1 \in \mathcal{I}$, Let $a_{\langle k_1, \ldots, k_n \rangle} = p_{k_1}$ and let $m = \mathbf{c}(k_1)$, the code of the C-set containing p_{k_1}. Then,

$$f_d(m) = k_1 \Leftrightarrow g_d(\langle k_1, \ldots, k_n \rangle) = \mathsf{tt}.$$

Since only one element of the C-set with index m can approximate d, $f_d(m)$ can be calculated by finding element a_k in C-set C_m such that $g_d(k)$ is true. If such an element exists, a dovetailing search will eventually find it. ∎

5.2 Computability in SPCF

To apply the preceding general framework to SPCF, we need to define an effective presentation δ^τ for each of the domains $\mathcal{T}[\![t]\!]$ of the SPCF model. We proceed by induction on the structure of the type t.

Conventions. In this subsection, the letters I, J range over indices of *finite* elements, i, j, k, l over indices of prime elements, and m, n over the indices of C-sets. A C-set in the domain $\mathcal{T}[\![\tau]\!]$ is called a c^τ-set. ∎

The design of an enumerator $\mathbf{E}^o : \mathbb{N}_\perp^E \to \mathbb{N}$ for the base domain is straightforward:

$$\begin{array}{rcll} \mathbf{E}^o(\perp) & = & 0 & \\ \mathbf{E}^o(\mathsf{error}_1) & = & 1 & \\ \mathbf{E}^o(\mathsf{error}_2) & = & 2 & \\ \mathbf{E}^o(x) & = & x+3 & \qquad (x \in \mathbb{N}) \end{array}$$

It is easy to confirm that the enumerator $\mathbf{E}^o$ is topologically sorted. Moreover, it is effective because the functions $\mathbf{c}^o$, $\mathbf{pr}^o$, and $\mathbf{err}^o$ are definable as primitive recursive functions:

$$\begin{array}{rcl} \mathbf{c}^o(0) & = & 0 \\ \mathbf{c}^o(x+1) & = & 1 \\ \mathbf{pr}^o(x) & = & 0 \end{array} \qquad \mathbf{err}^o(x) = \begin{cases} x & \text{if } x = 1 \text{ or } x = 2 \\ 0 & \text{otherwise} \end{cases}$$

The definition of the enumerator $\mathbf{E}^{\sigma\to\tau} : \mathcal{T}[\![\sigma \to \tau]\!] \to \mathbb{N}$ and corresponding functions $\mathbf{c}^{\sigma\to\tau}$, $\mathbf{pr}^{\sigma\to\tau}$, and $\mathbf{err}^{\sigma\to\tau}$ for higher types $\sigma \to \tau$ proceeds by sub-induction on the structure of prime functions using the prime approximation ordering. First, set $\mathbf{E}^{\sigma\to\tau}(\perp_{\sigma\to\tau}) = 0$. Next, let f be a prime function with predecessor f', *i.e.*, $f' \prec f$, and let l be the index of f'. Then,

$$\mathbf{E}^{\sigma\to\tau}(f) = \begin{cases} \langle\langle \mathsf{O}, I, k\rangle, l\rangle & \text{if } f \text{ is an output prime, determined by } f', a_I, \text{ and } p_k \\ \langle\langle \mathsf{S}, \langle I, m\rangle, n\rangle, l\rangle & \text{if } f \text{ is a schedule prime with } si_f(a_I, C_m) = C_n \end{cases}$$

where the tags O (for output prime) and S (for schedule prime) stand for 0 and 1, respectively. The functions $\mathbf{c}^{\sigma\to\tau}$, $\mathbf{pr}^{\sigma\to\tau}$ and $\mathbf{err}^{\sigma\to\tau}$ are defined as follows:

$$\begin{array}{rcl} \mathbf{c}^{\sigma\to\tau}(\langle\langle \mathsf{O}, I, k\rangle, l\rangle) & = & \langle\langle I, \mathbf{c}^\tau(k)\rangle, l\rangle \\ \mathbf{c}^{\sigma\to\tau}(\langle\langle \mathsf{S}, \langle I, m\rangle, n\rangle, l\rangle) & = & \langle\langle I, m\rangle, l\rangle \\ \mathbf{pr}^{\sigma\to\tau}(\langle\langle \cdots\rangle, l\rangle) & = & l \end{array} \qquad \begin{array}{rcl} \mathbf{err}^{\sigma\to\tau}(\langle\langle \mathsf{O}, I, k\rangle, l\rangle) & = & \mathbf{err}^\tau(k) \\ \mathbf{err}^{\sigma\to\tau}(\langle\langle \mathsf{S}, \langle I, m\rangle, n\rangle, l\rangle) & = & 0 \end{array}$$

Since $\langle 0, 0\rangle = 0$, $\mathbf{c}(0) = 0$ and $\mathbf{pr}(0) = 0$ as required. It is straightforward to verify that these functions are legitimate, *i.e.*, follow the conditions of an effective presentation.

Given the preceding definition of computability in the SPCF model, we can now state and prove the central claim of this section: *SPCF is a computationally complete language for its obserably sequential model.*

Theorem 5.8 *Let τ be an SPCF type. For every computable element $d \in \mathcal{T}[\![\tau]\!]$ there exists a closed SPCF term M such that $\mathcal{M}[\![M]\!] = d$.*

Proof Sketch. Let d be a computable element with characteristic function f_d. Since SPCF is a conservative (upward-compatible) extension of PCF, which can express all recursive functions in $\mathbb{N} \multimap \mathbb{N}$, SPCF contains a term N_d that computes the function in $f_d^\dagger$ corresponding to f_d. More precisely, $f_d^\dagger : \mathbb{N}_\perp^E \to_{os} \mathbb{N}_\perp^E$ satisfies the conditions:

$$\begin{array}{rcl} f_d^\dagger(x) & = & x \text{ iff } x \in \mathbb{E}_\perp \\ f_d^\dagger(m) & = & n \text{ iff } f_d(m) = n \\ f_d^\dagger(m) & = & \perp \text{ if } f_d(m) \text{ is undefined.} \end{array}$$

We call $f_d^\dagger$ the SPCF-characteristic function for d.

By the same argument, the functions **c**, **pr**, and **err** from any effective domain presentation have SPCF counterparts **C**, **Pr**, and **Err** corresponding to **c**, **pr**, and **err**.

To prove the theorem, all we have to do is construct an SPCF function Decode^τ that maps $f_d^\dagger$ to the corresponding element d. Unfortunately, the obvious approach to writing Decode^τ fails for higher types τ because it requires writing an inverse function Encode^τ, which maps an element d to the corresponding SPCF characteristic function $f_d^\dagger$. The latter function cannot be written in SPCF because *it is not observably sequential.* The function Encode^τ recognizes and discards rather than propagates error elements.

We can circumvent this problem by introducing an alternate functional representation for elements of D. Given an element $d \in D$, the *OS-characteristic* function f_d^* is defined in terms of the the characteristic function f_d for d as follows:

$$
\begin{array}{lcll}
f_d^*(x) & = & x & \text{for } x \in \mathbb{E}_\perp \\
f_d^*(m) & = & k & \text{iff } f_d(m) = k \text{ and } \mathbf{err}(k) = 0 \\
f_d^*(m) & = & \mathsf{error}_i & \text{iff } \exists\, m' \leq' m : \mathbf{err}(f_d(m')) = i \text{ for } i > 0 \\
f_d^*(m) & = & \perp & \text{otherwise.}
\end{array}
$$

$\mathsf{convert}^\tau \;= \lambda f.\lambda m.$
$\quad\quad\quad\quad\mathbf{letrec}\; z{=}\lambda m'.\,\mathbf{let}\; v{=}(f\; m')$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\mathbf{in}\; (\mathsf{if0}\;\; (\mathit{generr}^\tau\;\; v)$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad(\mathsf{if0}\;\;\; (=\;\; m'\; m)\; v$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad(z\;\, (\mathit{NextCset}\;\; m'\;\; m)))$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\Omega)$
$\quad\quad\quad\quad\;\mathbf{in}\; (z\;\, (\mathit{InitCset}^\tau\;\; m))$

$\mathit{generr}^\tau \;= \lambda v.\,\mathbf{let}\; \mathit{errnum}{=}(\mathbf{Err}^\tau\;\; v)$
$\quad\quad\quad\quad\;\mathbf{in}\; (\mathsf{if0}\;\; (\mathsf{sub1}\; \mathit{errnum})\; \mathsf{error}_1\; (\mathsf{if0}\; (\mathsf{sub1}^2\; \mathit{errnum})\; \mathsf{error}_2\; \ulcorner 0 \urcorner))$

Comment The function $(\mathit{NextCset}\;\, m'\;\, m)$ calculates the unique n such that $m' \prec' n \leq' m$.

$\mathit{NextCset}^\tau = \lambda m', m.(\mathsf{if0}\;\; (=\;\; (\mathbf{C}^\tau\;\; (\mathsf{2nd}\;\, m))\;\, m')$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad m$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{NextCset}^\tau\;\; m'\;\, (\mathbf{C}^\tau\;\; (\mathsf{2nd}\;\, m))))$

$\mathit{InitCset}^\tau = (\mathit{NextCset}^\tau\;\, \ulcorner 0 \urcorner)$

FIGURE 4: The Function $\mathsf{convert}$

Figure 4 contains the code of a procedure $\mathsf{convert}$ that converts a characteristic function $f_d^\dagger$ to a *OS*-characteristic function f_d^*, *i.e.*, ($\mathsf{convert}$ N_d) denotes f_d^*. Given an index m for a C-set, ($\mathsf{convert}$ f), the code for the *OS*-characteristic function, works as follows. It visits all C-sets that approximate m, starting at the lowest possible one, and checks whether f maps the current C-set index to an error-generating prime. If so, it generates the correct error value; if not, it outputs the prime index $(f\ m)$. The correctness of the function depends on correctness of the auxiliary function *NextCset* that gives unique n for a given m' and m such that $m' \prec' n \leq' m$. The correctness of the function *NextCset* can easily be verified by studying the encoding of the C-sets.

In contrast to Encode^τ, the function encode^τ mapping an element $d \in D$ to f_d^* is definable in SPCF. Similarly, the function decode^τ mapping a *OS*-characteristic function f_d^* to the corresponding element $d \in D$ is definable in SPCF. If we construct a procedure decode^τ, the proof of the theorem is complete, because

$$\mathcal{M}[\![(\mathsf{decode}^\tau\ (\mathsf{convert}\ x))]\!] = \mathcal{M}[\![(\mathsf{Decode}^\tau\ x)]\!]\,.$$

The function decode relies on the function encode; both these functions are simultaneously constructed inductively. The construction of the code for decode^τ and encode^τ proceeds by induction on the type τ. At each stage in the construction, we will assume the following induction hypothesis for smaller types, which simply states that encode and decode satisfy their specifications.

Induction Hypothesis: For every type τ:

1. There is an SPCF function decode^τ, which decodes a *OS*-characteristic function f_d^* into the data object d, *i.e.*,

$$\mathcal{M}[\![\mathsf{decode}^\tau]\!](f_d^*) = d\,.$$

2. There is an SPCF function encode^τ, which encodes a data object $e \in \mathcal{T}[\![\tau]\!]$ as its *OS*-characteristic function f_e^*, *i.e.*,

$$\mathcal{M}[\![\mathsf{encode}^\tau]\!](e) = f_e^*\,.$$

Consequently, we must show that the induction hypothesis holds for the constructed functions decode^τ and encode^τ.

The definition of the functions decode^o and encode^o for the base type o is straightforward:

$$\begin{aligned}\mathsf{decode}^o &= \lambda f_d^*.(f_d^*\ 1)\\ \mathsf{encode}^o &= \lambda x.\lambda n.(\mathsf{if0}\ n\ \ulcorner 0\urcorner\ (\mathsf{if0}\ (\mathsf{sub1}\ n)\ (+\ \ulcorner 3\urcorner\ x)\ \Omega))\end{aligned}$$

The induction hypothesis obviously holds for type o.

Assume that the induction hypothesis holds for the types σ and τ, implying that we can use the functions decode^σ, encode^σ, decode^τ, and encode^τ in the construction of the functions $\mathsf{decode}^{\sigma\to\tau}$ and $\mathsf{encode}^{\sigma\to\tau}$. The function $\mathsf{decode}^{\sigma\to\tau}$ is shown in Figure 5. It accepts an *OS*-characteristic function f_d^* and some argument e for the function d. It relies on the function encode^σ to encode e as its *OS*-characteristic function f_e^*. In the body of $\mathsf{decode}^{\sigma\to\tau}$, the auxiliary function *Apply* takes the arguments f_d^* and f_e^* constructs the *OS*-characteristic function $f_{d(e)}^*$ for $d(e)$. Then $\mathsf{decode}^{\sigma\to\tau}$ decodes the *OS*-characteristic function $f_{d(e)}^*$ to produce the data object $d(e)$.

To prove that $\mathsf{decode}^{\sigma\to\tau}$ satisfies the induction hypothesis, we observe that by the extensionality of $\mathcal{T}[\![\sigma\to\tau]\!]$

$$\mathcal{M}[\![\mathsf{decode}^{\sigma\to\tau}]\!](f_d^*) = d$$

$$
\begin{array}{ll}
\mathsf{decode}^{\sigma\to\tau} & = \lambda f_d^*.\lambda e^\sigma.(\mathsf{decode}^\tau\ (Apply^\tau\ f_d^*\ (\mathsf{encode}^\sigma\ e))) \\[1em]
Apply^\tau & = \lambda f_d^*, f_e^*.\lambda m\,. \\
& \quad \textbf{letrec } z = \lambda l\,.(\textbf{let } \mathit{final} = (\mathsf{1st}\ l) \\
& \qquad\qquad\qquad \textbf{in } (\mathsf{if0}\ \ (=\ O\ (\mathsf{1st}\ \mathit{final})) \\
& \qquad\qquad\qquad\qquad (\textbf{let } \langle I, k\rangle = (\mathsf{2nd}\ \mathit{final}) \\
& \qquad\qquad\qquad\qquad\quad \textbf{in } (\mathsf{if0}\ \ (=\ m\ (\mathbf{C}^\tau\ k)) \\
& \qquad\qquad\qquad\qquad\qquad\quad k \\
& \qquad\qquad\qquad\qquad\qquad\quad (\textbf{let*}\ n_\tau = (NextCset^\tau\ (\mathbf{C}^\tau\ k)\ m) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad l = (f_d^*\ \langle\langle I, n\rangle, l\rangle) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \ (z\ l)))) \\
& \qquad\qquad\qquad\qquad (\textbf{let*}\ \langle\langle I, m'\rangle, n\rangle = (\mathsf{2nd}\ \mathit{final}) \\
& \qquad\qquad\qquad\qquad\qquad I' = (\mathsf{sort}\ \langle (f_e^*\ n), I\rangle) \\
& \qquad\qquad\qquad\qquad\qquad l = (f_d^*\ \langle\langle I', m'\rangle, l\rangle) \\
& \qquad\qquad\qquad\qquad\quad \textbf{in } \ (z\ l)))) \\
& \quad\ \ \textbf{in } (z\ \ulcorner 0 \urcorner)
\end{array}
$$

FIGURE 5: The Function *decode*

iff for all finite elements $e \in \mathcal{T}[\![\sigma]\!]$,

$$(\mathcal{M}[\![\mathsf{decode}^{\sigma\to\tau}]\!](f_d^*))(e) = d(e).$$

By the definition of $\mathsf{decode}^{\sigma\to\tau}$, the preceding line reduces to showing

$$\mathcal{M}[\![(\mathsf{decode}^\tau(Apply\ F_d\ (\mathsf{encode}^\sigma\ E)))]\!] = d(e)$$

where the terms F_d and E have the denotations f_d^* and e respectively. The term E exists by the induction hypothesis. If we assume $\mathcal{M}[\![Apply]\!](f_d^*)(f_e^*) = f_{d(e)}^*$ for all d and e, then the proof obligation reduces to

$$\mathcal{M}[\![\mathsf{decode}^\tau]\!](f_{d(e)}^*) = d(e),$$

which holds by the induction hypothesis. Thus, we have reduced the claim about the correctness of decode function to a specific lemma about the function *Apply*.

Lemma 5.9 *Let f_d^*, f_e^* be the OS-characteristic functions for $d \in \mathcal{T}[\![\sigma \to \tau]\!]$ and $e \in \mathcal{T}[\![s]\!]$, respectively. Then,*

$$\mathcal{M}[\![Apply]\!](f_d^*)(f_e^*) = f_{d(e)}^*.$$

Proof Sketch. Set $h = \mathcal{M}[\![Apply]\!](f_d^*)(f_e^*)$. The lemma holds if h is the OS-characteristic function for $d(e)$, *i.e.*

$$\forall\, m, i :\ h(m) = i \Leftrightarrow p_i \in d(e) \text{ and } \mathbf{c}^\tau(i) = m\,.$$

We show that $h(m)$ gives the correct result by analyzing the three possible cases:

$f^*_{d(e)}(m) = j$. We must show that $h(m) = j$. First, we observe that for any $d \in \mathcal{T}[\![\sigma \to \tau]\!]$, and $e \in \mathcal{T}[\![\sigma]\!]$, $p_j \in d(e)$ iff there exists a least prime $q_* \in d$ such that $q_*(e) = p_j$. Otherwise, d would not have a unique sequentiality index at each point during its exploration of e in the direction $\mathbf{c}^\tau(j)$.

By simple rewriting rules $h(m) = z(0)$ and q_0 approximates q_*. By the following claim, $z(0) = j$, proving the required result for this case.

Claim: $z(l) = j$ if q_l approximates q_*.

We prove the claim by induction on the distance (see Definition 3.4) from q_l to q_*.

Base Case: If the distance is 0 then q_l must be equal to q_*. Then, by simple rewriting rules, the function z generates j.

Inductive Case: The next approximation to q_* is constructed from current approximation l as follows:

- If l is the index of an output prime $\langle\langle \mathsf{O}, I, k\rangle, l'\rangle$, the function d has two options:
  - it can output a prime $p_{k'}$ in the C-set m' immediately above p_k in the direction C^τ_m without further examining the input e; or
  - it can examine the input e above a_I to help determine which prime to output in the C-set m'.

  In either case, the next element above l in d in the direction of q_* must be in the C-set $\langle\langle I, m'\rangle, l\rangle$. Therefore the index of the next approximation to q_* is $f^*_d(\langle\langle I, m'\rangle, l\rangle)$.
- If l is a schedule prime $\langle\langle \mathsf{S}, \langle I, m'\rangle, n\rangle, l'\rangle$, then if $\mathsf{error}_i^{C_n} \in e$ we must have $\mathsf{error}_i^{m'} \in d(e)$. In other words, the function d has to explore the C-set n in the input e where C_n covers a_I. Since q_* exists, this exploration yields a better approximation $a_{I'} = a_I \sqcup r_i$ to the input e where $r_i \in e$ and $\mathbf{c}^\sigma(i) = n$. Therefore, the next prime immediately above l in the direction of q_* (if it exists) is a prime in the C-set $\langle\langle I', m'\rangle, l\rangle$.

Therefore the next approximation decreases the distance from q_* by 1. By the induction hypothesis, z applied to this better approximation gives j as the result.

$f^*_{d(e)}(m) = \mathsf{error}_i$. Let m' be the C-set approximating m and $\mathsf{error}_i^{C_{m'}} \in d(e)$. Then q_* be the maximum prime in d such that $q_*(e) \prec C_{m'}$.

By rewriting the code, $h(m) = z(0)$. Since q_0 approximates q_*, the following claim gives $h(m) = \mathsf{error}_i$.

Claim: If q_l approximates q_* then $z(l) = \mathsf{error}_i$.

The proof of the claim proceeds by induction on the distance from q_l to q_*.

Base Case: If $q_l = q_*$, then the function h has two choices depending on q_l. If it is an output prime, it will generate an error without looking for further input. If it is a schedule prime, it will generate an error while looking at an error in the input e. In either case the corresponding error value error_i is generated.

Inductive Case: As in the preceding case, each time z reduces the distance between q_* and q_l by 1. Therefore, by the induction hypothesis, $z(l) = \mathsf{error}_i$.

$f^*_{d(e)}(m) = \bot$. There are two possibilities:

- $\exists p_{j'} \in d(e) \cap C^{\tau}_{m'}$ such that $m' \leq' m$ and $p_{j'}$ is not in the direction C^{τ}_{m}. In other words, there is a conflicting prime along the direction C^{τ}_{m}.
- $\exists p_{j'} \in d(e)$ such that $p_{j'} \notin C^{\tau}_{m}$ and it is the maximum prime in the direction C^{τ}_{m}.

In either case, divergence of the function $h(m)$ can be proved by an analysis of the behavior of z as in the preceding cases.

Thus, in each of the above cases, $h(m)$ returns the same result as $f^*_{d(e)}$. That is, h is the *OS*-characteristic function for $d(e)$. Hence, we proved the claim $\mathcal{M}[\![Apply]\!](f^*_d)(f^*_e) = f^*_{d(e)}$. $\blacksquare_{5.10}$

The remaining obligation in the proof of Theorem 5.8 is to show that the family of functions $\mathsf{encode}^{\sigma\to\tau}$ satisfy the condition specified in the induction hypothesis. The definition of the functions $\mathsf{encode}^{\sigma\to\tau}$ is given in Figure 6.

For the moment, let us assume that the auxiliary function *chk* satisfies its specification: $chk(g, m) = f^*_g(m)$ if $\exists p_k \in g : p_k \prec C^{\sigma\to\tau}_m$. We can prove that $\mathsf{encode}^{\sigma\to\tau}$ returns f^*_g for a data object $g \in \mathcal{T}[\![\sigma \to \tau]\!]$ as follows.

First, we observe that *InitCset* is an auxiliary function that returns the index of first C-set above 0 that approximates the given input. If the input is 0 it returns 0. Therefore the result of $InitCset(m)$ always approximates m and there is no non-zero C-set that approximates the $InitCset(m)$.

Set $h = \mathcal{M}[\![\mathsf{encode}]\!](g)$. The function encode satisfies the induction hypothesis if $h = f^*_g$. We prove this equality by analyzing the possible results of $f^*_d(m)$:

$f^*_g(m) \in \mathbb{N}$: Therefore, we must prove that $h(m) = f^*_g(m)$. By simple rewriting rules, $h(m) = z(InitCset(m))$. If we prove the following claim, we have the result that $h(m) = f^*_g(m)$ since $InitCset(m) \leq' m$.

Claim: If $m' \leq' m$, then $z(m') = f^*_g(m)$.

The proof of the claim is by induction on the distance between the C-sets (see Definition 3.4) $C_{m'}$ and C_m.

Base Case: If the distance is 0 then $m' = m$. Since $\exists p_k \in g : p_k \prec C_{m'}$, the function $chk(g, m)$ returns $f^*_g(m)$ by Lemma 5.10. Therefore, $z(m') = f^*_g(m)$.

Inductive Case: By the assumption that $f^*_g(m) \in \mathbb{N}$, the function $chk(g, m')$ returns a prime in the direction C_m. Therefore, z is recursively invoked with $NextCset(m', m)$, the index of the C-set one step closer to m. By induction hypothesis and simple rewriting, $z(m') = z(NextCset(m', m)) = f^*_g(m)$.

$f^*_g(m) = \mathsf{error}_i$: We must prove that $h(m)$ also generates the error element error_i, for $i = 1, 2$. As in the preceding case, $h(m) = z(InitCset(m))$.

Since $f^*_g(m) = \mathsf{error}_i$ there must be a least $n \leq' m$ where $f^*_g(n) = \mathsf{error}_i$. Since $InitCset(m) \leq' n$, we can prove the required result for this case by proving the following claim.

Claim: If $m' \leq' n$, then $z(m') = \mathsf{error}_i$.

Proof of the claim proceeds by induction on the distance between m' and n. It is similar to the preceding case.

$f_g^*(m) = \bot$: We must prove that $h(m) = \bot$. Then only of the statements is true:

- There is a p_k in g such that and p_k is not in the direction C_m and $p_k \in C_n$ where $n \leq' m$.
- There is a p_k in g such that p_k is the maximum prime in the direction C_m. Let C_n be the C-set covering p_k. It is easy to see $n \leq' m$.

In either case, the claim that $m' \leq' n$ implies $z(m') = \bot$ can be proved by induction on the distance from m' and n. Therefore $h(m) = z(\mathit{InitCset}(m)) = \bot$.

Therefore, $f_g^*(m) = h(m)$ for all valid values of m, implying that encode satisfies its specification. Hence, we have reduced the proof of Theorem 5.8 to the proof of the following lemma.

Lemma 5.10 *Let $g \in \mathcal{T}[\![\sigma \to \tau]\!]$ and m be the* C*-set code in the domain $\mathcal{T}[\![\sigma \to \tau]\!]$. If there is a $p_k \in g$ such that p_k is covered by $C_m^{\sigma\to\tau}$, then $chk(g, m) = f_g^*(m)$.*

Proof Sketch. Let g and m be the inputs to the function chk and let $m = \langle I, n, j\rangle$. Then $p_j \in \mathcal{T}[\![\sigma \to \tau]\!]$ approximates g by assumption. In addition, by the definition of our coding scheme, C_n covers $p_j(a_I)$.

We first observe that chk, with the help of *Extend*, generates a *OS*-characteristic function $f_{a'}^*$. This function agrees with $f_{a_I}^*$ on the queries m' such that $\mathbf{c}^\sigma(j) = m$ and $p_j \in a_I$; but, if the function $f_{a'}^*$ is explored beyond a_I, it "throws" an index back to chk, which returns this index as the final result.

The specifications for the auxiliary functions *GenFun*, *Direction*, *InElt* and $<?$ is given in the comments. It is straightforward to verify that the functions follow the specifications by the induction on encodings of the primes and C-sets.

Now we analyze the four possible cases of $f_g^*(m)$:

$f_g^*(m) = \bot$: In this case, we must show that $chk(g, m) = \bot$. Since $m = \langle I, n, j\rangle$, we know that $f_g^*(m)$ is $\bot$ iff $g(a_I) \cap C_n^\tau$ is empty. In other words, there is no prime element in the C-set C_n^τ at the output of $g(a_I)$. Therefore, by the induction hypothesis of encode, $\mathsf{encode}^\tau(g(a_I))(n)$ should diverge. Since $a_I = a'$ for all the queries within a_I, the function chk diverges on the given inputs g and m.

$f_g^*(m) = \mathsf{error}_i$: Just as in the preceding case, $g(a_I)$ should have an error prime, $\mathsf{error}_i^{C_n}$. By the induction hypothesis of encode, $\mathsf{encode}^\tau(g(a_I))(n)$ returns error_i; therefore the result of the function chk is error_i.

$f_g^*(m) = \langle\langle \mathsf{O}, I, l\rangle, j\rangle$: In that case, $g(a_I)$ should have the prime element p_l at the C-set C_n^τ. By the induction hypothesis of encode, $\mathsf{encode}^\tau(g(a_I))(n) = l$. Therefore, the function chk returns the index $\langle\langle \mathsf{O}, I, l\rangle, j\rangle$. Notice that by the assumption that $p_k \prec C_m^{\sigma\to\tau}$ is in g, all of the input a_I is necessary for generating the output at the C-set C_n^τ, which means that the index of the output prime is correct.

$f^*_g(m) = \langle\langle \mathsf{S}, \langle I, n\rangle, n'\rangle, j\rangle$: Therefore, $\mathsf{error}_i^{C_n} \in g(a \sqcup \mathsf{error}_i^{C_{n'}})$. That means, the function g checks the input at the C-set $C^\sigma_{n'}$ to generate the output at the C-set C^τ_n. When the function $f^*_{a'}$ is queried for the input at the C-set $C^\sigma_{n'}$ it throws the index $\langle\langle \mathsf{S}, \langle I, n\rangle, n'\rangle, j\rangle$, which is returned by the function *chk*.

Therefore, in all possible cases, the function $chk(g, m)$ returns $f^*_g(m)$ assuming that the prime covered by m is in g. $\blacksquare_{5.11}$

This also finishes the sketch of the theorem's proof. $\blacksquare$

Acknowledgments. We are grateful to Albert Meyer for raising the completeness question of the $\lambda\beta\eta$ calculus addressed in Section 4 and to John Gateley and Jon Riecke for discussions about the material in that section. Discussions with Pierre-Louis Curien improved our understanding of the sequential model and brought the Kahn-Plotkin-Winskel material to our attention.

References

1. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* Revised Edition. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.
2. Berry, G. Séquentialité de l'evaluation formelle des λ-expressions. In *Proc. 3rd International Colloquium on Programming*, 1978.
3. Berry, G. *Modèles complètement adéquats et stables des lambda-calculus typé.* Ph.D. dissertation, Université Paris VII, 1979.
4. Berry, G. and P-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.* **20**, 1982, 265–321.
5. Berry, G. and P-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language cds. In *Algebraic Methods in Semantics*, edited by J. Reynolds and M.Nivat. Cambridge University Press. London, 1985, 35–88.
6. Berry, G., P-L. Curien, and P.-P. Lévy. Full-abstraction of sequential languages: the state of the art. In *Algebraic Methods in Semantics*, edited by J. Reynolds and M.Nivat. Cambridge University Press. London, 1985, 89–131.
7. Cartwright, R. Lambda: The ultimate combinator. *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy.* Edited by Vladimir Lifschitz (ed.), Academic Press, 1991, 27–46.
8. Cartwright, R. and A. Demers. The topology of program termination. In *Proc. Symposium on Logic in Computer Science*, 1988, 296–308.
9. Cartwright, R.S. and M. Felleisen. Observable sequentiality and full abstraction. Technical Report 91-167. Rice University Department of Computer Science, 1991. Preliminary version: In *Proc. 19th ACM Symposium on Principles of Programming Languages*, 1992, 328–342.
10. Cartwright, R.S., Curien, P.-L., and M. Felleisen. Fully abstract models of observably sequential languages. Forthcoming, December 1992.

11. Curien, P-L. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, London. 1986. Birkhäuser, Revised Edition, to appear.
12. Curien, P.-L.. Observable algorithms on concrete data structures. In *Proc 7th Symposium on Logic in Computer Science*, 1992, to appear.
13. Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming* **17**, 1991, 35–75. Preliminary version in: *Proc. 3rd European Symposium on Programming.* Neil Jones, Ed. Lecture Notes in Computer Science, 432. Springer Verlag, Berlin, 1990, 134–151.
14. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. *Theor. Comput. Sci.* **102**, 1992.
15. Felleisen, M., D.P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**(3), 1987, 205–237. Preliminary version in: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.
16. Friedman, H. Equality between functionals. In *Logic Colloquium'73*, Rohit Parikh (Ed.), Lecture Notes in Mathematics 453, Springer Verlag, Berlin, 1973, 22–37.
17. Kahn, G. and G. Plotkin. Structures des donnés concrètes (Domaines Concrètes). IRIA Report 336. 1978.
18. Meyer, A.R. What is a model of the λ-calculus? *Inf. and Control* **52**, 1982, 87–122.
19. Milner, R. Fully abstract models of typed λ-calculi. *Theor. Comput. Sci.* **4**, 1977, 1–22.
20. Mulmuley, K. *Full Abstraction and Semantic Equivalences.* Ph.D. dissertation, Carnegie Mellon University, 1985. MIT Press, Cambridge, Massachusetts, 1986.
21. Plotkin, G. Notes on completeness of the full continuous type hierarchy. Unpublished manuscript, MIT, December 1982.
22. Plotkin, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.
23. Riecke, J. Statman's 1-section theorem. Tech. Rpt. MS-CIS-92-03. Uinversity of Pennsylvania. 1992.
24. Riecke, J. and R. Subrahmanyam. Algebraic reasoning and completeness in typed languages. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, 1992, to appear.
25. Sazonov, V.Y. Expressibility of functions in D.Scott's LCF language. *Algebra i Logika* **15**(3), 1976, 308–330.
26. Scott, D. S. Domains for denotational semantics. In *Proc. International Conference on on Automata, Languages, and Programming*, Lecture Notes in Mathematics 140, Springer-Verlag, Berlin, 1982, ??–??.
27. Scott, D.S. Data types as lattices. *SIAM J. Comput.* **5**(3), 1976, 522–587.
28. Scott, D.S. *Lectures on a Mathematical Theory of Computation.* Techn. Monograph PRG-19, Oxford University Computing Laboratory, Programming Research Group, 1981.

29. Sitaram, D. and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990, 161–175.

30. Statman, R. Equality between functionals revisited. In L.A. Harrinton, et al (Eds), *Harvey Friedman's Research on the Foundations of Mathematics.* Studies in Logic. North-Holland, 1985, 331–338.

31. Statman, R. Completeness, invariance, and lambda-definability. *Journal of Symbolic Logic* **47**, 1982, 17–26.

32. Vuillemin, J. Proof techniques for recursive programs. IRIA Report. 1973.

33. Winskel, G. *Events in Computations.* Ph.D. dissertation, University of Edinburgh, 1980.

$$
\begin{array}{ll}
\mathsf{encode}^{\sigma\to\tau} & = \lambda g.\lambda m. \\
& \quad \textbf{letrec } z{=}\lambda m'.(\mathsf{if0}\ \ (=\ m'\ m)\ (chk^{\sigma\to\tau}\ g\ m) \\
& \qquad\qquad\qquad\qquad (\mathsf{if0}\ (<?^{\tau}\ (chk^{\sigma\to\tau}\ g\ m')\ m) \\
& \qquad\qquad\qquad\qquad\quad (z\ (NextCset^{\sigma\to\tau}\ m'\ m)) \\
& \qquad\qquad\qquad\qquad\quad \Omega)) \\
& \quad\ \ \textbf{in}\ \ (z\ (InitCset\ m))
\end{array}
$$

$$
\begin{array}{ll}
chk^{\sigma\to\tau} & = \lambda g, m.\ (\textbf{let}\ \ \langle\langle I, n\rangle, j\rangle =\ m \\
& \qquad\quad \textbf{in}\ \ \mathsf{call/cdc}(\lambda t.(\textbf{let}{*}\ \ f^*_{a'} = (Extend^{\sigma}\ I\ n\ j\ t) \\
& \qquad\qquad\qquad\qquad\qquad\quad\ \ a' \ = (\mathsf{decode}^{\sigma}\ f^*_{a'}) \\
& \qquad\qquad\qquad\qquad\qquad\quad\ \ l \ \ = ((\mathsf{encode}^{\tau}\ (g\ a'))\ n) \\
& \qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \langle\langle \mathsf{O}, I, l\rangle, j\rangle))))
\end{array}
$$

$$
\begin{array}{ll}
Extend^{\sigma} & = \lambda I, n, j, t.\lambda m'.(\mathsf{if0}\ \ (InElt^{\sigma}\ I\ m') \\
& \qquad\qquad\qquad\qquad ((GenFun^{\sigma}\ I)\ m') \\
& \qquad\qquad\qquad\qquad (t\ \langle\langle \mathsf{S}, \langle i, n\rangle, (Direction^{\sigma}\ I\ m')\rangle\ j\rangle))
\end{array}
$$

Comment $(GenFun\ I)$ generates $f^*_{a_I}$.

$$
\begin{array}{ll}
GenFun^{\sigma} & = \lambda I.\,\lambda m.\ (\textbf{letrec } z{=}\lambda I.(\mathsf{if0}\ \ (=\ (\mathbf{C}^{\sigma}\ (\mathsf{1st}\ I))\ m) \\
& \qquad\qquad\qquad\qquad\qquad\quad (\mathsf{1st}\ I) \\
& \qquad\qquad\qquad\qquad\qquad\quad (z\ (\mathsf{2nd}\ I))) \\
& \qquad\quad\ \textbf{in}\ (z\ I))
\end{array}
$$

Comment $(Direction\ I\ m) = n$ where a_I is covered by C_n and $n \leq' m$.

$$
\begin{array}{ll}
Direction^{\sigma} & = \lambda I, m.(\mathsf{if0}\ I \\
& \qquad\qquad (InitCset\ m) \\
& \qquad\qquad (\mathsf{if0}\ (<?^{\sigma}\ (\mathsf{1st}\ I)\ m) \\
& \qquad\qquad\quad (NextCset^{\sigma}\ (\mathbf{C}^{\sigma}\ (\mathsf{1st}\ I))\ m) \\
& \qquad\qquad\quad (Direction^{\sigma}\ (\mathsf{2nd}\ I)\ m)))
\end{array}
$$

Comment $(InElt\ I\ m) = \ulcorner 0 \urcorner$ iff $p_k \in a_I$ and $\mathbf{c}(k) = m$; otherwise $\ulcorner 1 \urcorner$.

$$
\begin{array}{ll}
InElt^{\sigma} & = \lambda I, m.(\mathsf{if0}\ \ m\ \ \ulcorner 0 \urcorner \\
& \qquad\qquad (\mathsf{if0}\ \ I\ \ulcorner 1 \urcorner \\
& \qquad\qquad\quad (\mathsf{if0}\ (=\ \ (\mathbf{C}^{\sigma}\ (\mathsf{1st}\ I))\ m) \\
& \qquad\qquad\qquad \ulcorner 0 \urcorner \\
& \qquad\qquad\qquad (InElt^{\sigma}\ (\mathsf{2nd}\ I)\ m))))
\end{array}
$$

Comment $(<?\ k\ m) = \ulcorner 0 \urcorner$ iff $p_k < C_m$; otherwise $\ulcorner 1 \urcorner$.

$$
\begin{array}{ll}
<?^{\tau} & = \lambda k, m.(\mathsf{if0}\ m\ \ulcorner 1 \urcorner \\
& \qquad\qquad (\mathsf{if0}\ (=\ \ k\ (\mathsf{2nd}\ m)) \\
& \qquad\qquad\quad \ulcorner 0 \urcorner \\
& \qquad\qquad\quad (<?^{\tau}\ k\ (\mathbf{C}^{\tau}\ (\mathsf{2nd}\ m)))))
\end{array}
$$

FIGURE 6: The Function *encode*