FULLY ABSTRACT SEMANTICS FOR OBSERVABLY SEQUENTIAL LANGUAGES

Robert Cartwright Pierre-Louis Curien Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251-1892

Also appeared in *Information and Computation 1994*
Rice University Technical Report # 93-219
Minor revisions: October 2004

Fully Abstract Semantics for Observably Sequential Languages

Robert Cartwright*
Department of Computer Science, Rice University, Houston, TX 77005

Pierre-Louis Curien†
Ecole Normale Supérieure, LIENS-CNRS, 45 rue d'Ulm, 75230 Paris, France

Matthias Felleisen‡
Department of Computer Science, Rice University, Houston, TX 77005

Abstract

One of the major challenges in denotational semantics is the construction of a fully abstract semantics for a higher-order *sequential* programming language. For the past fifteen years, research on this problem has focused on developing a semantics for PCF, an idealized functional programming language based on the typed λ-calculus. Unlike most practical languages, PCF has no facilities for *observing* and *exploiting* the evaluation order of arguments to procedures. Since we believe that these facilities play a crucial role in sequential computation, this paper focuses on a sequential extension of PCF, called SPCF, that includes two classes of control operators: a possibly empty set of error generators and a collection of *catch* and *throw* constructs. For each set of error generators, the paper presents a fully abstract semantics for SPCF. If the set of error generators is empty, the semantics interprets all procedures—including catch and throw—as Berry-Curien *sequential algorithms*. If the language contains error generators, procedures denote *manifestly sequential* functions. The manifestly sequential functions form a Scott domain that is isomorphic to a domain of decision trees, which is the natural extension of the Berry-Curien domain of sequential algorithms in the presence of errors.

1 Full Abstraction and Sequentiality

A denotational semantics for a programming language determines two natural equivalence relations on program phrases. The first relation, *denotational equivalence*, equates two phrases if and only if they have the same meaning (denotation). The second relation, *observational equivalence*, equates two phrases if and only if they have the same "observable

*Supported in part by NSF grant CCR 91-22518.
†Supported in part by Esprit BRA CLICS, and a visit at DEC SRC, Palo-Alto.
‡Supported in part by NSF grants CCR 89-17022, CCR 91-22518, and a visit at Carnegie Mellon University.

behavior". In the denotational framework, observable behavior refers to the output produced by *entire* programs. Thus, two phrases are *observationally equivalent* if and only if they can be interchanged in an *arbitrary* program without affecting the output.

Observational equivalence is the fundamental constraint governing program optimization. Since the users of a program are primarily interested in its input-output behavior, a compiler can optimize a program by interchanging observationally equivalent phrases. Unfortunately, it is difficult to prove that two phrases are observationally equivalent because the proof must address all possible program contexts. Denotational equivalence is more tractable because it can be analyzed using the algebraic and topological structure of denotations.

As a result, one of the primary goals of programming language theory is the construction of denotational semantics for practical languages where denotational equivalence and observational equivalence coincide. A semantics with this property is called *fully abstract*.

Denotational equivalence implies observational equivalence because a denotational semantics is compositional. But the converse rarely holds for *practical* programming languages; some observationally equivalent phrases inevitably have different meanings. The following example shows why a conventional continuous function semantics is not fully abstract for higher-order sequential languages. Consider the family of procedures[1] p_i defined in a sequential, call-by-name functional language L:

$$\begin{array}{lll} p_i(f) = \textbf{if} & f(\Omega,\mathsf{false}) & \textbf{then}\ \Omega \\ \ \textbf{else if} & f(\mathsf{false},\Omega) & \textbf{then}\ \Omega \\ \ \textbf{else if} & f(\mathsf{true},\mathsf{true}) & \textbf{then}\ i\ \ \textbf{else}\ \Omega\ . \end{array}$$

In this equation, Ω denotes any divergent phrase and *i* denotes an arbitrary natural number. By a proof technique due to Plotkin [24], p_i is observationally equivalent to the procedure p defined by $p(f) = \Omega$. In essence, the procedures p_i diverge for all inputs because the only possible inputs f are *sequential* procedures.

Nevertheless, the conventional semantics for language L includes functions that evaluate their arguments in parallel. A simple example of a parallel function is *parallel-and*, which returns false if *either* input is false and returns true if *both* inputs are true. The application of p_i to *parallel-and* yields the answer *i* instead of $\perp$ because divergent subcomputations are ignored by *parallel-and*. Consequently, the conventional semantics for language L fails to identify p_i and *p*.

Essentially the same example can be constructed in any practical deterministic programming language where procedures can be passed as parameters [21, 30]. For example, in call-by-value languages, the procedures p_i can be rewritten so that the parameter f takes constant procedures as arguments and uses these procedures to simulate call-by-name boolean arguments [30, 29]. Indeed, all commonly used deterministic languages are *sequential*.

In rough terms, a language is sequential if it can be implemented without time-slicing among multiple threads of control. The technical usage of the term *sequential* in this paper should not be confused with the informal uses of term in the context of parallel processing. In informal usage, languages are classified as "sequential" or "parallel" based on the

[1] We use the term *procedure* rather than *function* to refer to the primitive operators in a functional programming language because procedures are not necessarily interpreted as functions by a semantics.

intended implementation strategy rather than the semantic properties of the language. In fact, practical parallel languages are either sequential in the technical sense or they are nondeterministic. Deterministic parallel languages like C*, *Lisp, and some parallel dialects of Fortran support parallelism in program execution through "data-parallel" operations or "hierarchical" parallelism where every task executed in parallel must terminate for the entire computation to terminate. The functions computable using these two forms of parallelism are sequential.

Parallel functions like *parallel-and* add to the expressiveness of deterministic languages, but the price paid in execution efficiency for using these operations is exorbitant. Moreover, the additional expressiveness is not useful in practice, because it applies only to computations that are unbounded. For this reason, parallel functions have not been included in any practical language.

1.1 History of Previous Work

Milner [22], Plotkin [24], and Sazonov [26] were the first researchers to study the full abstraction problem for sequential languages. They focused on constructing a fully abstract semantics for PCF, a call-by-name functional language based on the typed λ-calculus. Plotkin and Sazonov observed that the continuous function semantics for PCF is not fully abstract using essentially the same counterexample as given above. Milner and Plotkin developed different strategies for eliminating the discrepancy between the continuous function semantics and the observable behavior of PCF.

Milner eliminated parallel functions from the semantics by constructing Scott domains from equivalence classes of observationally equivalent terms. Milner showed that his semantics is fully abstract, and that it is unique up to isomorphism among fully abstract, extensional semantics. But Milner's construction of a semantics is not regarded as "denotational" because it fails to identify any algebraic structure within programs. Plotkin extended PCF by adding *parallel* deterministic operations, eliminating the discrepancy between the continuous function semantics and the procedures definable in the language. Unfortunately, neither Milner's nor Plotkin's result showed how to construct a fully abstract denotational semantics for a *sequential* language like PCF.

In a subsequent investigation of sequential languages, Berry and Curien [3, 4, 6, 8] constructed a semantics for PCF with more restrictive domains for procedure denotations. Berry introduced the stable function space construction, which eliminated many—but not all—parallel functions.[2] To address this problem and to produce a semantic characterization of sequentiality, Berry and Curien [4, 5] proposed interpreting procedures as *sequential algorithms over concrete domains* [18]. Roughly speaking, a concrete domain is a domain that is isomorphic to a domain consisting of potentially infinite trees. A sequential algorithm is a continuous function *plus* a strategy for evaluating its arguments. While this approach eliminates all parallel functions, the resulting semantics is not extensional because it contains different sequential algorithms that compute the same function. In addition, PCF cannot express all of the observations that characterize sequential algorithms, such as the

[2]Indeed, it even introduces new distinctions, which are impossible in the continuous function space construction. This observation is due to Jim and Meyer [16]; for details on this observation, we refer to their paper.

order of argument evaluation. As a result, the sequential algorithm semantics for PCF is not fully abstract.

Berry and Curien did show, however, that the concrete domains and sequential algorithms form a Cartesian-closed category and they developed a language called CDS0 for which the sequential algorithm semantics is fully abstract. CDS0 differs from conventional functional languages like PCF because it uses sequential algorithms—represented as decision trees—rather than PCF-like procedures as higher order data objects. Hence, higher order data objects in CDS0 are not extensional.

Mulmuley [23] generalized Milner's work by showing how to construct a fully abstract semantics for PCF as a quotient of a conventional semantics based on *lattices* instead of *cpos*. He defined a retraction on the conventional semantics that equates all parallel functions with the "overdefined" element top ($\top$). Jung and Stoughton [17] recently extended Mulmuley's work by directly constructing a projection from the inductively reachable subalgebra of the continuous function semantics to the fully abstract semantics for PCF. However, like Milner's original fully abstract semantics, both Mulmuley's and Jung and Stoughton's semantics are "syntactic" in flavor because they rely on a quotient construction based on observational equivalence and definability. For more details on the history of the full abstraction problem for sequential languages, we refer the reader to three extensive surveys [6, 9, 32].

1.2 Summary of Results

This paper integrates recent work by Cartwright and Felleisen [7] and by Curien [10] on fully abstract semantics for simple, sequential extensions of PCF. It solves an important form of the full abstraction problem, namely the construction of a fully abstract semantics for a realistic functional language extending PCF. The key insight underlying this work is the invention of a new form of function space: the domain of *manifestly sequential* functions. The manifestly sequential functions form a projective subspace [19] of the continuous functions—provided that the domain and codomain meet certain topological constraints.

By drawing on their intuitions as programmers, Cartwright and Felleisen [7] invented the manifestly sequential functions to construct a fully abstract denotational semantics for a *sequential* extension of PCF called SPCF. Curien [10] recognized that the *manifestly sequential functions* were an extensional refinement of the *sequential algorithms* that he and Berry had developed in an attempt to construct a fully abstract semantics for PCF. His paper showed that the framework of "concrete data structure" (*cds*) domains can easily be extended to accommodate manifestly sequential functions without invalidating any of the framework's important properties.[3]

This paper presents a new framework for defining the category of manifestly sequential domains and functions called *sequential data structures*. The paper subsumes both original presentations by the authors. In the framework of sequential data structures, manifestly

[3]The term *manifestly sequential* was coined by the authors in the process of writing this paper. Cartwright and Felleisen [7] defined the manifestly sequential functions in the context of two error elements. They called these functions *error-sensitive* functions. *Manifest sequentiality* is a generalization of the original notion of error-sensitivity that is applicable to domains with a single error element.

sequential functions are represented as decision trees, formalized as sets of paths. Sequential data structures are similar to the concrete data structures that Berry and Curien used to define sequential algorithms, but they are simpler and make the tree structure of elements explicit.

The paper is organized as follows. Section 2 contains the mathematical preliminaries for the remainder of the paper. Section 3 describes the syntax and semantics of PCF. In Section 4, we compare PCF to realistic functional languages. Unlike practical functional languages (Common Lisp, Scheme), PCF lacks *control operators* for generating errors and performing non-local exits, which makes it impossible to observe the order in which procedures evaluate their arguments. Extending PCF to include these facilities naturally leads to a fully abstract semantics using manifestly sequential functions represented by decision trees. Section 5 introduces suitable domains and functions for constructing such a semantics and Section 6 contains the proof that this collection of domains and functions forms an appropriate category. Section 7 is devoted to the proof of full abstraction for SPCF. Finally, Section 8 presents an operational semantics and an adequacy theorem for SPCF. The adequacy proof uses a strong normalization argument for bounded recursion, simplifying Curien's original proof [10]. The appendix relates the sequential data structures presented in this paper to the original concrete data structures used by Berry and Curien.

2 Mathematical Preliminaries

Before we present the syntax and semantics of PCF and SPCF, we need to introduce some definitions and notation.

Sets. We use standard mathematical notation and terminology for sets and functions. In particular, $\mathbb{N}$ denotes the set of natural numbers. The indices $i, \ldots, n$ range over a (subset of) $\mathbb{N}$. The expression $\underline{\lambda}x. \cdots x \cdots$ denotes the mathematical function f defined by the equation $f(x) = \cdots x \cdots$, where the domain and codomain (range) of f are determined by context. If A and B are sets, then the equations

$$\begin{array}{rcl} A \setminus B & = & \{a \,|\, a \in A, a \notin B\} \\ A \uplus B & = & \{\langle a, 1\rangle, \langle b, 2\rangle \,|\, a \in A, b \in B\} \end{array}$$

define the *set difference* $A \setminus B$ between A and B and the *discriminated union* $A \uplus B$ of the sets A and B.

Partial Orders and Domains. A *partial order* is a pair $(D, \sqsubseteq)$ where D is a set and $\sqsubseteq$ is a binary relation on D that is reflexive, transitive, and antisymmetric. In contexts where no confusion can arise, we abuse notation and use the symbol D to refer to the partial order. If $x \sqsubseteq y$, we say *x approximates y* or *y dominates x*. If $x \sqsubseteq y$ and $x \neq y$, then we write $x \sqsubset y$ and say *x* is *strictly below y*.

An *upper bound* of $X \subseteq D$ is an element of D that dominates all elements $x \in X$. A *least upper bound* of X is an upper bound that approximates all other upper bounds. Since the partial order D is antisymmetric, a least upper bound of X is unique; it is denoted $\bigsqcup X$.

The least upper bound of a pair x, y of elements in D is denoted $x \sqcup y$. The notions of *lower bound* and *greatest lower bound* are defined similarly; $\sqcap X$ is the greatest lower bound of $X \subseteq D$ and $x \sqcap y$ is the greatest lower bound of the pair $x, y \in D$. A subset $X \subseteq D$ is *bounded* if it has an upper bound. If a pair $x, y \in D$ is bounded, we say it is *consistent* and write $x \uparrow y$. A *non-empty* subset X of a partial order D is *directed* if every pair of elements from X has an upper bound in X.

A *complete* partial order (cpo) is a partial order $(D, \sqsubseteq)$ such that D has a least element $\perp$ and every directed subset X of D has a least upper bound. An element y of a cpo D is *finite* if for every directed $X \subseteq D$ such that $y \sqsubseteq \bigsqcup X$, there exists an element $x \in X$ such that $y \sqsubseteq x$.

A cpo is *flat* if $x \sqsubseteq y$ if and only if $x = \perp$ or $x = y$. A cpo is *consistently-complete* if every bounded pair of elements has a least upper bound. A cpo D is *algebraic* if for every $x \in D$ the set of finite elements below x is directed and its least upper bound is x:

$$x = \bigsqcup \{d \sqsubseteq x \mid d \text{ is finite }\}.$$

It is ω-*algebraic* if it is algebraic and the set of finite elements is countable.

A function $f : D \longrightarrow D'$ for cpos $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ is *continuous* if it preserves limits of directed sets:

$$f(\bigsqcup X) = \bigsqcup \{f(x) \mid x \in X\}$$

for any directed set X. A function $f : D_1 \times \ldots \times D_n \longrightarrow D$ is strict in argument position k, $1 \leq k \leq n$, if for all $x_1 \in D_1, \ldots, x_{k-1} \in D_{k-1}, x_{k+1} \in D_{k+1}, \ldots, x_n \in D_n$:

$$f(x_1, \ldots, x_{k-1}, \perp_k, x_{k+1}, \ldots, x_n) = \perp\ .$$

A *Scott domain* (or simply *domain*) is an ω-algebraic, consistently-complete cpo. $\mathbb{N}_\perp$ is the flat domain of natural numbers with least element $\perp$.

Paths. If Σ is an arbitrary set, called the *alphabet*, then Σ^* denotes the set of finite *paths* over Σ.[4] The empty path is denoted ϵ.

Every path $p \in \Sigma^*$ can be interpreted as a *partial* function from $\mathbb{N}$ to Σ that satisfies the following two conditions:

- There exists $n \in \mathbb{N}$ such that $p(n)$ is undefined.
- If $p(i+1)$ is defined, then $p(i)$ is defined.

We often write $p@i$ instead of $p(i)$; this notation corresponds to interpreting p as a sequence of tokens

$$p@0 \cdot p@1 \cdot \ldots \cdot p@n$$

where $\cdot$ is a separator delimiting the boundaries between tokens. The notation $|p|$ denotes the length of a path, *i.e.*, $|p| = n + 1$ if $p@n$ is defined and $p@(n+1)$ is undefined. It is also convenient to identify the elements of Σ with the paths of length 1.

[4] In the literature, a path is usually called a *word* or a *list*, but the reader will see that for our purposes it is more appropriate to use the terminology "path".

Σ^* forms a partial order under prefix ordering $\sqsubseteq$ defined by the rule: $q \sqsubseteq p$ if $q \subseteq p$ as partial functions. In other words, $q \sqsubseteq p$ when p and q satisfy the following condition: if $q(n)$ is defined, then $p(n)$ is defined and $p(n) = q(n)$.

If $p \in \Sigma^*$ is a path and a a member of the alphabet, then $p \cdot a$ is the path that extends p with a: $(p \cdot a)@|p| = a$. Similarly, $a \cdot p$ is the path that shifts p to the right and places a at the beginning: $(a \cdot p)@i + 1 = p@i$ for all $i > 0$ and $(a \cdot p)@0 = a$. The symbol $\cdot$ also stands for path concatenation. Since paths over Σ are not tokens, this convention does not cause any ambiguity.

Given two subsets, Φ and Υ, of an alphabet Σ, such that $\Phi \cup \Upsilon = \Sigma$, the set of *alternating* paths, denoted by $(\Phi, \Upsilon)^*$, is the set of paths p in Σ^* that satisfy the following conditions:

1. if $p@(2n + 1)$ is defined, then $p@(2n + 1) \in \Upsilon$; and
2. if $p@2n$ is defined, then $p@2n \in \Phi$.

In particular, $\epsilon \in (\Phi, \Upsilon)^*$. An alternating path p is called non-repetitive in Φ if it satisfies the following additional condition:

3. if $p@i, p@j$ are in Φ and $p@i = p@j$ then $i = j$.

It is non-repetitive in Υ if it satisfies the same condition relative to Υ.

Categories. A *category* $\mathcal{C}$ is 4-tuple $\langle Obj_{\mathcal{C}}, \longrightarrow_{\mathcal{C}}, \circ, \mathsf{id}_{\mathcal{C}} \rangle$ where

1. $Obj_{\mathcal{C}}$ is a collection of objects;
2. $\longrightarrow_{\mathcal{C}}$ is a family of collections $A \longrightarrow_{\mathcal{C}} B$ of arrows called *homsets*, one for each pair of objects $A, B \in Obj_{\mathcal{C}}$;
3. $\circ$ is a composition operation on arrows $(\bigcup \longrightarrow_{\mathcal{C}})$ such that (i) $f \circ g$ is an arrow in $A \longrightarrow_{\mathcal{C}} C$ if f is in $B \longrightarrow_{\mathcal{C}} C$ and g is in $A \longrightarrow_{\mathcal{C}} B$; and (ii) the operation is associative:
$$(f \circ g) \circ h = f \circ (g \circ h)\,;$$
4. $\mathsf{id}_{\mathcal{C}}$ is a collection of arrows $\mathsf{id}^A \in A \longrightarrow_{\mathcal{C}} A$, one for each object $A \in Obj_{\mathcal{C}}$ such that id^A is the identity with respect to the composition operation:
$$\mathsf{id}^A \circ f = f \circ \mathsf{id}^A = f.$$

When no confusion can arise, subscripts and superscripts are dropped in the notation for categories.

An object 1 in a category $\mathcal{C}$ is *terminal* if there exists exactly one arrow 1^A in $A \longrightarrow 1$ for each object A. The object $A_1 \times A_2$ is a *product* of the two objects A_1, A_2 if

1. there exist arrows π_i in $A_1 \times A_2 \longrightarrow A_i$ for $i = 1, 2$; and

2. for any object B and any arrows f_1 in $B \longrightarrow A_1$, f_2 in $B \longrightarrow A_2$, there exists an arrow $\langle f_1, f_2 \rangle$ in $B \longrightarrow A_1 \times A_2$, the *pair* of f_1 and f_2, such that

$$\begin{array}{rcl} \pi_1 \circ \langle f_1, f_2 \rangle & = & f_1 \\ \pi_2 \circ \langle f_1, f_2 \rangle & = & f_2 \\ \langle \pi_1 \circ f, \pi_2 \circ f \rangle & = & f \ . \end{array}$$

   The last equation forces the arrow $\langle f_1, f_2 \rangle$ to be unique.

In the sequel, we will work with *multiple* products of the shape $1 \times A_n \times \ldots \times A_1, n > 0$, with corresponding arrows $\pi_i^n \in 1 \times A_n \times \ldots \times A_1 \longrightarrow A_i, 1 \leq i \leq n$ such that:

$$\begin{array}{rcl} 1 \times A_n \times \ldots \times A_2 \times A_1 & = & (\ldots (1 \times A_n) \times \ldots) \times A_1 \\ \pi_i^n & = & \pi_2 \circ \underbrace{\pi_1 \circ \ldots \circ \pi_1}_{i-1} . \end{array}$$

A category $\mathcal{C}$ is *cartesian* if (i) it has a terminal object and (ii) a product object for any two objects.

An object $[A \Rightarrow B]$ in a cartesian category $\mathcal{C}$ is the *exponent* of the objects A and B if for any object C:

1. there is a bijection Λ from $C \times A \longrightarrow B$ to $C \longrightarrow [A \Rightarrow B]$ with inverse Λ^{-1} such that for all f in $C \times A \longrightarrow B$ and g in $C \longrightarrow [A \Rightarrow B]$:

$$\begin{array}{rcl} \Lambda^{-1}(\Lambda(f)) & = & f \\ \Lambda(\Lambda^{-1}(g)) & = & g \end{array}$$

2. for any object D and any two arrows f in $C \times A \longrightarrow B$ and g in $D \longrightarrow C$:

$$\Lambda(f) \circ g = \Lambda(f \circ \langle g \circ \pi_1, \pi_2 \rangle) \, .$$

The bijection Λ is called the *currying* operator. The Λ-inverse of the identity arrow for the exponent object $A \Rightarrow B$ is called the *application arrow*:

$$App = \Lambda^{-1}(\mathsf{id}^{A \Rightarrow B}).$$

It is easy to show that the preceding conditions on Λ are equivalent to the following two equations:

$$\begin{array}{rcl} f & = & \Lambda^{-1}(\mathsf{id}) \circ \langle \Lambda(f) \circ \pi_1, \pi_2 \rangle = App \circ \langle \Lambda(f) \circ \pi_1, \pi_2 \rangle \\ f & = & \Lambda(\Lambda^{-1}(\mathsf{id}) \circ \langle f \circ \pi_1, \pi_2 \rangle) = \Lambda(App \circ \langle f \circ \pi_1, \pi_2 \rangle) \, . \end{array}$$

A cartesian category is *cartesian-closed* if it has exponents for any two objects. The category $\mathcal{D}om$, consisting of (Scott) domains as objects and continuous functions as arrows, is cartesian-closed.

A cartesian-closed category is *cpo-enriched* if:

- the collection of arrows (homset) $A \longrightarrow B$ between every two objects A and B forms a cpo with respect to a supplementary approximation ordering $\sqsubseteq_{A \longrightarrow B}$;
- arrow composition $\circ$ is continuous with respect to the approximation ordering on arrows; and
- the currying constructor $\Lambda(\cdot)$ and the pairing constructor $\langle \cdot, \cdot \rangle$ are monotonic with respect to the approximation ordering on arrows.

The continuity of function composition implies the continuity of the pairing constructor and currying operator. The category $\mathcal{D}om$ is cpo-enriched.

3 PCF: Syntax and Semantics

PCF is a functional language consisting of the typed λ-calculus augmented by numerals, procedures to increment and decrement numbers, a conditional procedure, and a family of fixed-point operators. The first two subsections of this section introduce the syntax and semantics of PCF. The last subsection introduces the notions of full abstraction and sequentiality.

3.1 PCF Syntax

The set of PCF types consists of a single ground type (o), denoting the set of *observable* values, and an infinite collection of procedure types ($\sigma \to \tau$), denoting sets of procedural values.

Definition 3.1 (*PCF Types*) The following grammar generates the set of PCF types:[5]

$$\tau ::= o \mid (\tau \to \tau)\,.$$

The meta-variables $\tau, \tau_1, \tau', \nu, \sigma, \ldots$ range over PCF types; $\tau_1 \to \tau_2 \to \ldots \to \tau_n$ abbreviates $\tau_1 \to (\tau_2 \to \ldots \to \tau_n)$. It is easy to show that all types τ have the shape

$$\tau_1 \to \tau_2 \to \ldots \to \tau_n \to o$$

for some types $\tau_1, \ldots, \tau_n$. The *arity* of a type τ with shape

$$\tau_1 \to \tau_2 \to \ldots \to \tau_n \to o$$

is n. ∎

An unchecked PCF phrase M is either a numeral ($\ulcorner n \urcorner$ for $n \geq 0$), a typed variable (x^τ), a λ-abstraction, or one of two types of applications.

[5] We follow the usual practice of using meta-variables for sets as if they were members of a set and vice versa.

Definition 3.2 (*Unchecked PCF Phrases***)** The following grammar generates the unchecked PCF phrases:

$$\begin{array}{rcl} M & ::= & c \mid x \mid (\lambda x.M) \mid (M\ M) \mid (f\ M) \\ c & ::= & \Omega \mid \ulcorner 0 \urcorner \mid \ulcorner 1 \urcorner \mid \ \ldots \\ f & ::= & \mathsf{add1} \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{Y}^{\tau} \\ x & ::= & x^{\tau} \mid y^{\tau} \mid \ \ldots\ . \end{array}$$

The meta-variables $L, M, N, L', M', N', L_1, M_1, N_1, \ldots$ range over phrases. ∎

As is customary in functional languages, λ-abstraction is the only binding form in PCF: $\lambda x.M$ binds x in M. A variable that is not bound by a surrounding λ-abstraction is *free*. A phrase in which all variables are bound is *closed*. Phrases are identified if they are interconvertible by consistent renaming of bound variables, *e.g.*, $\lambda x.x \equiv \lambda y.y$. Thus, an assertion stating that $\lambda x.M$ has the property "*x* is not a member of some finite set of variables" means that $\lambda x.M$ is an appropriate member of the equivalence class. The description of the operational semantics requires the standard notion of capture-free substitution: $M[N/x]$ denotes the substitution of all free occurrences of *x* in *M* by *N*. Finally, the standard syntactic conventions of the λ-calculus apply, *e.g.*, $\lambda xyz.M$ abbreviates $\lambda x.(\lambda y.(\lambda z.M))$ and $(M\ N\ L)$ abbreviates $((M\ N)\ L)$.

Many important semantic properties of languages rely on notion of a *context*, which is a program phrase with holes. The set of contexts is generated by an extension of the grammar for unchecked PCF phrases.

Definition 3.3 (*Contexts***)** The language of PCF contexts is generated by the grammar for the set of unchecked PCF phrases augmented by the production:

$$M ::= [\ ]_0 \mid [\ ]_1 \ldots$$

where the symbol $[\ ]_i, i \in \mathbb{N}$ is called a hole.

The notation $C[\ ]$ designates an arbitrary context with a single hole $[\ ]$. Filling all occurrences of the hole $[\ ]$ in the context $C[\ ]$ with a phrase *M* yields the phrase $C[M]$ in which λ-abstractions in *C* may capture free variables in *M*. A context with m disjoint sets of holes, written:

$$C\underbrace{[\ ], \ldots, [\ ]}_{m},$$

can be filled with *m* terms: $C[M_1, \ldots, M_m]$.

Filling a context with a context, written as $C[C'[\ ]]$ yields another context. ∎

A (typed) PCF phrase is an unchecked PCF phrase that also conforms to the type constraints imposed by the typed λ-calculus. Every (typed) PCF phrase M has a unique type τ. Each variable x^{σ} has type σ. Similarly, each constant c in the language has a designated type τ_c. Within a (typed) PCF phrase, the types of the phrases in each application $(M\ N)$ must match: M must have a procedure type $\sigma \to \tau$ and N must have type σ. The type of the application $(M\ N)$ is τ. Each λ-abstraction $(\lambda x^{\sigma}.M)$ has type $\sigma \to \tau$ where τ is the type of M.

Definition 3.4 (*PCF Phrases, Programs, Program Contexts***)** Let M be an unchecked PCF phrase and let A be a *list* of typed variables, called a *type environment*, that contains the free variables of M. The relation "A proves that M is a *PCF phrase* of type τ," written $A \vdash M : \tau$, is the least relation generated by the following axioms and inference rules:

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

$$A \vdash \ulcorner n \urcorner : o \qquad \frac{A \vdash M : o}{A \vdash (\mathsf{add1}\ M) : o} \qquad \frac{A \vdash M : o}{A \vdash (\mathsf{sub1}\ M) : o} \quad .$$

$$A \vdash \Omega : o \qquad \frac{A \vdash M : o}{A \vdash (\mathsf{if0}\ M) : o \to o \to o} \qquad \frac{A \vdash M : \tau \to \tau}{A \vdash (\mathsf{Y}^\tau\ M) : \tau}$$

If $A \vdash M : \tau$ holds, it is called a *typing* for M. Since the choice of A does not affect the type inference process, we often just say M has type τ. The significance of the type environment A is explained in the definition of the semantics of PCF.

A (typed) *PCF phrase* is an unchecked PCF phrase that has some type τ. A *PCF program* is a closed PCF phrase of type o. The hole of a context $C[\ ]$ has type τ if $C[x^\tau]$ is a typed phrase. Given the PCF phrases M and N of type τ, a *program context* $C[\ ]$ for M and N is a context such that $C[M]$ and $C[N]$ are programs.

A PCF phrase M of type τ has arity *n* if τ has arity *n*. ∎

The semantics of PCF phrases is easy to explain informally. Numerals denote numbers, Ω represents a diverging computation of type o, λ-abstractions denote call-by-name procedures, juxtapositions denote procedure applications, and the procedural constants $\mathsf{add1}$, $\mathsf{sub1}$, $\mathsf{if0}$, and Y^σ have their usual meanings. Hence, the procedures $\mathsf{add1}$ and $\mathsf{sub1}$ increment and decrement their integer inputs, respectively; the latter diverges on 0. The procedure $\mathsf{if0}$ evaluates its first argument: if the result is 0, it evaluates the second argument and returns it; otherwise it evaluates the third argument and returns it. The Y^σ operator computes the fixed-points of procedures of type $\sigma \to \sigma$.

The following phrases frequently appear in the analysis of PCF and warrant special notation:

$$\begin{array}{rcl} \Omega^o & = & \Omega \\ \Omega^{\sigma \to \tau} & = & \lambda x^\sigma . \Omega^\tau \\ \mathsf{Y}_0^\sigma & = & \lambda x^{\sigma \longrightarrow \sigma} . \Omega^\sigma \\ \mathsf{Y}_{n+1}^\sigma & = & \lambda f^{\sigma \longrightarrow \sigma} . (f\ (\mathsf{Y}_n^\sigma\ f)) . \end{array}$$

The phrase Ω^τ is a divergent phrase of type o. The phrase Y_i^σ approximates the Y^σ combinator: for f of type $\sigma \to \sigma$, $\mathsf{Y}_i^\sigma f$ is the i-fold application of f to Ω^σ. Roughly speaking, Y^σ is the least upper bound of $\{\mathsf{Y}_i^\sigma \mid i \in \mathbb{N}\}$.

3.2 PCF Semantics

A denotational semantics for a language based on the typed λ-calculus assigns meaning to types and phrases by mapping them to appropriate mathematical entities. These entities form a cartesian-closed cpo-enriched category that interprets the divergent term Ω^σ as the least element in the corresponding domain of arrows and interprets the fixed-point combinator Y^σ as the least upper bound of the approximations Y^σ_i.

Definition 3.5 (*Semantics for languages based on the typed λ-calculus***)** Let L be a language consisting of the typed λ-calculus augmented by a set of constants including the divergent constant Ω, and the Y^σ-combinator for each type σ. A *semantics* for L is determined by a triple $\langle \mathcal{C}, \mathcal{C}^o, \mathcal{C}[\![\ \cdot\ ]\!]\rangle$ where $\mathcal{C}$ is cartesian-closed cpo-enriched category, $\mathcal{C}^o$ is an object in $\mathcal{C}$, and $\mathcal{C}[\![\ \cdot\ ]\!]$ is a function mapping each constant c of type τ in L, *except* Y^τ, to an arrow in $1 \longrightarrow \mathcal{C}^\tau$ such that the following conditions are satisfied:

- The homset $\perp^{1\longrightarrow\mathcal{C}^o}$ is a flat Scott domain.
- Ω is interpreted as the least element $\perp^{1\longrightarrow\mathcal{C}^o}$ in the Scott domain $1 \longrightarrow \mathcal{C}^o$:
$$\mathcal{C}[\![\,\Omega\,]\!] = \perp^{1\longrightarrow\mathcal{C}^o} .$$
- For any three objects A, B and C:
$$\begin{array}{rcl} \Lambda(\perp^{C\times A\longrightarrow B}) & = & \perp^{C\longrightarrow[A\Rightarrow B]} \\ \langle \perp^{A\longrightarrow B}, \perp^{A\longrightarrow C}\rangle & = & \perp^{A\longrightarrow B\times C} . \end{array}$$

Given $\mathcal{C}^o$, the family of objects $\mathcal{C}^\tau$ in the category $\mathcal{C}$ is inductively defined by the equation:
$$\mathcal{C}^{\sigma\rightarrow\tau} = \mathcal{C}^\sigma \Rightarrow \mathcal{C}^\tau .$$

The meaning function $\mathcal{C}[\![\ \cdot\ ]\!]$ is extended from constants to typed phrases excluding Y^τ as follows. Let $A \vdash N : \tau$ be (the conclusion of) a proof that N is a typed phrase for some type environment $A = x_n^{\tau_n}, \ldots, x_1^{\tau_1}, n \geq 0$. Then the *meaning* of N is an arrow in
$$1 \times \mathcal{C}^{\tau_n} \times \ldots \times \mathcal{C}^{\tau_1} \longrightarrow \mathcal{C}^\tau$$
inductively defined by the following equations:
$$\begin{array}{rcll} \mathcal{C}[\![\,A \vdash c : \tau\,]\!] & = & \mathcal{C}[\![\,c\,]\!] \circ 1^{1\times\mathcal{C}^{\tau_n}\times\ldots\times\mathcal{C}^{\tau_1}} & \\ \mathcal{C}[\![\,A \vdash x_i^\tau : \tau\,]\!] & = & \pi_i^n & \\ \mathcal{C}[\![\,A \vdash \lambda x^\tau\,.\,M : \tau'\,]\!] & = & \Lambda(\mathcal{C}[\![\,A, x^\tau \vdash M : \tau'\,]\!]) & x^\tau \text{ is not in } A \\ \mathcal{C}[\![\,A \vdash (M_1\ M_2) : \tau'\,]\!] & = & App \circ \langle \mathcal{C}[\![\,A \vdash M_1 : \tau \rightarrow \tau'\,]\!]\,, \mathcal{C}[\![\,A \vdash M_2 : \tau\,]\!]\rangle . & \end{array}$$

The type environment A plays a crucial role in the extension of $\mathcal{C}[\![\ \cdot\ ]\!]$ to typed phrases containing free variables, because it determines which projection function is used to select each variable x_i^τ.

Given the meaning function $\mathcal{C}[\![\ \cdot\ ]\!]$ for typed phrases excluding Y^τ, let the extension of $\mathcal{C}[\![\ \cdot\ ]\!]$ to Y^τ be defined by:
$$\mathcal{C}[\![\,A \vdash \mathsf{Y}^\tau : (\tau \rightarrow \tau) \rightarrow \tau\,]\!] \quad = \quad \bigsqcup \{\mathcal{C}[\![\,A \vdash \mathsf{Y}_n^\tau : (\tau \rightarrow \tau) \rightarrow \tau\,]\!] \mid n \in \mathbb{N}\} .$$

For the sake of notational economy, we refer to the semantics determined by the triple $\langle \mathcal{C}, \mathcal{C}^o, \mathcal{C}[\![\ \cdot\ ]\!]\rangle$ simply by the symbol $\mathcal{C}$. For closed typed terms M, we often abbreviate $\mathcal{C}[\![\, A \vdash M : \tau \,]\!]$ by $\mathcal{C}[\![\, A \vdash M \,]\!]$ or by $\mathcal{C}[\![\, \vdash M \,]\!]$, since the type of M can be reconstructed from the types of the constants that appear in M.

The *semantics of a program P* is $\mathcal{C}[\![\, \vdash P \,]\!]$, an arrow in $1 \longrightarrow \mathcal{C}^o$. ∎

A semantics validates the (β) and (fix) axioms of the typed λ-calculus with Y as well as its inference rules [8].

Lemma 3.6 *Let $\mathcal{C}$ be a semantics for PCF. Then for all PCF phrases M and N, contexts $C[\ \ ]$, appropriate type environments A, and variables x^τ:*

$$\mathcal{C}[\![\, A \vdash C[((\lambda x^\tau \,.\, M)\ N)] \,]\!] = \mathcal{C}[\![\, A \vdash C[M[N/x^\tau]] \,]\!]$$

and

$$\mathcal{C}[\![\, A \vdash C[(\mathsf{Y}\ M)] \,]\!] = \mathcal{C}[\![\, A \vdash C[(M\ (\mathsf{Y}M))] \,]\!]\ .$$

The conventional semantics for PCF based on continuous functions is an instance of the general definition. It is based on the cpo-enriched category $\mathcal{D}om$ whose objects are Scott domains and whose arrow sets $A \longrightarrow B$ are the sets of continuous functions from domain A to domain B.

Definition 3.7 (*$\mathcal{D}om$ Semantics of PCF, Adequate Semantics, PCF-like Languages*) The cpo-enriched cartesian-closed category of the conventional semantics for PCF is $\mathcal{D}om$. The ground type o is interpreted as $\mathbb{N}_\perp$:

$$\mathcal{D}om^o = \mathbb{N}_\perp .$$

The semantics assigns the following meaning to constants:

$$\begin{aligned}
\mathcal{D}om[\![\, \Omega^o \,]\!] &= \underline{\lambda}\rho : 1\ .\ \perp \\
\mathcal{D}om[\![\, \ulcorner n \urcorner \,]\!] &= \underline{\lambda}\rho : 1\ .\ n \\
\mathcal{D}om[\![\, \mathsf{add1} \,]\!] &= \underline{\lambda}\rho : 1\ .\ \underline{\lambda}m\ .\ \begin{cases} m+1 & m \in \mathbb{N} \\ \perp & m = \perp \end{cases} \\
\mathcal{D}om[\![\, \mathsf{sub1} \,]\!] &= \underline{\lambda}\rho : 1\ .\ \underline{\lambda}m\ .\ \begin{cases} m-1 & m > 0 \\ \perp & m \in \{\perp, 0\} \end{cases} \\
\mathcal{D}om[\![\, \mathsf{if0} \,]\!] &= \underline{\lambda}\rho : 1\ .\ \underline{\lambda}l\ .\ \underline{\lambda}m\ .\ \underline{\lambda}n\ .\ \begin{cases} m & l = 0 \\ n & l > 0 \\ \perp & l = \perp \end{cases}
\end{aligned}$$

where 1 in $\mathcal{D}om$ is the trivial, one-element domain.

A semantics for PCF that assigns the same meaning as the $\mathcal{D}om$ semantics to the ground type o and to all PCF programs is called *adequate*.

For any language L based on typed λ-calculus that extends PCF (differs from PCF only by extending the set of constants), a semantics for L is *PCF-like* if it assigns a flat superdomain of $\mathbb{N}_\perp$ to the base type and the same meaning as the $\mathcal{D}om$ semantics to PCF programs. ∎

3.3 Full Abstraction and Sequentiality

Given the preceding definition of a PCF-like language, we are finally ready to give a rigorous definition of denotational and observational equivalence, the two equivalence relations on typed phrases discussed at the beginning of the paper.

Definition 3.8 (*Denotational and Observational Equivalence and Approximation, Full Abstraction***)** In a PCF-like programming language with semantics $\mathcal{C}$, let M and M' be phrases that have the same type (*i.e.*, $A \vdash M : \tau$ and $A \vdash M' : \tau$ for some list of typed variables A). Then

- *M denotationally approximates M'*, written $M \sqsubseteq M'$ if
$$\mathcal{C}[\![\, A \vdash M : \tau \,]\!] \sqsubseteq \mathcal{C}[\![\, A \vdash M' : \tau \,]\!].$$
- *M is denotationally equivalent to M'*, written $M \equiv M'$, if
$$\mathcal{C}[\![\, A \vdash M : \tau \,]\!] = \mathcal{C}[\![\, A \vdash M' : \tau \,]\!].$$
- *M observationally approximates M'*, written $M \precsim M'$, if $\mathcal{C}$ assigns approximating meanings to the programs $C[M]$ and $C[M']$ for all program contexts $C[\ \ ]$ for M and M':
$$\mathcal{C}[\![\, \vdash C[M] \,]\!] \sqsubseteq \mathcal{C}[\![\, \vdash C[M'] \,]\!]\,.$$
- *M is observationally equivalent to M'*, written $M \simeq M'$, if $\mathcal{C}$ assigns the same meaning to the programs $C[M]$ and $C[M']$ for all program contexts $C[\ \ ]$ for M and M':
$$\mathcal{C}[\![\, \vdash C[M] \,]\!] = \mathcal{C}[\![\, \vdash C[M'] \,]\!].$$

A semantics $\mathcal{C}$ is (*equationally*) *fully abstract* if for all pairs of typed phrases M and M' of the same type, $M \simeq M'$ iff $M \equiv M'$; it is inequationally fully abstract if the proposition also holds for approximations: $M \precsim M'$ iff $M \sqsubseteq M'$.

In the presence of two different semantics $\mathcal{C}$ and $\mathcal{C}'$ for a language, we attach identifying subscripts to the denotational equivalence and approximation relations, *e.g.*, $\equiv_{\mathcal{C}}$, $\sqsubseteq_{\mathcal{C}'}$. ∎

Denotational and observational equivalence are different in character. While the former relation depends on the choice of the semantics, the latter does not as long as the semantics is adequate. More precisely, if $\mathcal{C}'$ is an adequate semantics for PCF distinct from $\mathcal{C}$, it still determines the same observational equivalence relation $\simeq$ but possibly a different denotational equivalence relation. The two relations are distinct in the $\mathcal{D}om$ semantics of PCF.

The lack of full abstraction for the $\mathcal{D}om$ semantics of PCF is due to the presence of *parallel* functions in the domain of continuous functions, *e.g.*, *parallel-and*. Yet, PCF can only define *sequential* functions, which evaluate their arguments in some specified order and diverge when they begin to evaluate a diverging argument. The procedures p_i defined in the introduction exploit this discrepancy: they behave differently only on inputs that are not sequential. Hence they are denotationally distinct and observationally equivalent.

Finally, we can precisely define what it means for the language to be *sequential*.

Definition 3.9 (*Sequential Language***)** A PCF-like language with semantics $\mathcal{C}$ is *sequential* if the following condition holds. Let C be a context with k holes of ground type and no free variables and $M_1, \ldots, M_k$ be closed phrases such that $\mathcal{C}[\![\, C[M_1, \ldots, M_k] \,]\!] \neq \perp$ and $\mathcal{C}[\![\, C[\Omega, \ldots, \Omega] \,]\!] = \perp$. Then there exists $j \in \mathbb{N}, 1 \leq j \leq k$, called a (*syntactic*) *sequentiality index* of C, such that

$$\mathcal{C}[\![\, C[M'_1, \ldots, M'_{j-1}, \Omega, M'_{j+1}, \ldots, M'_k] \,]\!] = \perp$$

for all phrases M'_i (of correct type). ∎

This definition is based on Vuillemin's definition [33] of sequentiality for first-order functions. Plotkin [24:Activity Lemma] and Berry [6:Theorem 3.6.4] proved that PCF is sequential, using an adequate[6] operational semantics for PCF. Plotkin's proof is direct, while Berry's proof is obtained as a corollary of a general syntactic sequentiality theorem [2, 6 : Theorem 3.6.2, 1 : Theorem 14.4.8].

We conclude the section with two useful examples of PCF procedures. The following two equations specify two versions of the binary addition procedure:

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

The first version $+_l$ recurs on the first argument x; the second version recurs on the second argument y. In the $\mathcal{D}om$ semantics of PCF, these two procedures denote exactly the same function, namely, binary addition. However, the two procedures clearly employ distinct strategies to inspect their arguments. Although this difference is not observable in PCF, it is observable in many practical languages. In the context of PCF, these two procedures can be distinguished only by inspecting the program text. But in more practical languages, control operators can be used to determine the evaluation order of procedures without inspecting their text. The next section focuses on the conflict between PCF and more practical languages.

4 Observing Sequentiality

As a pedagogic language, PCF lacks several constructs that are essential in practice. For example, PCF does not have any provision for generating run-time errors or handling exceptions. Although this design choice simplifies the definition of the language, we believe

[6] An operational semantics of a PCF-like language is *adequate* if the evaluator diverges on a program precisely when the program denotes bottom (see Definition 3.7).

that it has diverted people from discovering fully abstract semantics for simple, sequential extensions of PCF. In the following subsections, we explain why the addition of two simple control mechanisms to PCF is important from the perspective of language design and how their presence facilitates the construction of a fully abstract semantics.

4.1 Using Errors, Programmers Can Observe the Order of Evaluation

Since PCF excludes negative numbers, the predecessor operation for numbers diverges when applied to zero. While this design decision avoids the complication of generating and propagating error values in the semantics, it withholds important information from the programmer that is readily available in real implementations. A programmer who mistakenly tries to take the predecessor of zero would like to be informed of this fact instead of seeing his program diverge for the sake of a "simpler" semantics. The inclusion of error values in the language has a dramatic impact on the underlying semantics because it permits programmers to observe the evaluation order of subexpressions in programs by conducting simple experiments.

To report run-time errors in PCF, we can add *error constants*, $\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n$, $n > 1$, of ground type to the language and demand that all procedures in the extended language be *manifestly sequential*. An operation is manifestly sequential if it propagates any errors that it encounters in evaluating its arguments. In PCF with errors, a programmer can observe the evaluation order among subexpressions by exploiting the manifest sequentiality of all program operations. Specifically, let $C[M_1, \ldots, M_m]$ be a terminating PCF program such that $C[\Omega, \ldots, \Omega]$ diverges. To find out which of the *m* distinguished subexpressions is evaluated first, the programmer can evaluate the expression $C[\mathsf{e}_1, \ldots, \mathsf{e}_m]$ assuming m errors are available. The result e_i indicates which expression M_i was evaluated first. The same observation can be made with only two errors at the cost of conducting m experiments. Experiment i replaces M_j for $j \neq i$ by e_1 and M_i by e_2. The index of the experiment that yields the answer e_2 identifies which subexpression is evaluated first.

Using this form of experimentation, a programmer can distinguish procedures that denote the same function over $\mathbb{N}_\perp$ such as the addition procedures $+_l$ and $+_r$ defined in Section 3. The expression $(+_l\ \mathsf{e}_1\ \mathsf{e}_2)$ produces e_1 because $+_l$ evaluates its left argument first; $(+_r\ \mathsf{e}_1\ \mathsf{e}_2)$ returns e_2 because $+_r$ evaluates its right argument first.

4.2 Using Control Operators, Programs Can Observe the Order of Evaluation

In PCF with errors, the sequential behavior of procedures is observable *externally* by the programmer but this information is not accessible *internally* within programs. Consequently, a program cannot exploit the order of evaluation among the arguments in a procedure application. Since knowledge of the evaluation order can lead to shorter, more expressive, and faster programs, many practical languages include a non-local control operator that can propagate ordinary data values along the evaluation path that an error value would take.

The original version of Scheme [31], for example, contained a lexically-scoped CATCH construct for implementing non-local exits from expressions. Thus, the evaluation of the

$$
\begin{array}{l}
product = (\lambda l\,.\,(\textsc{Catch}\ e\ \ (\mathsf{Y}(\lambda \pi\,.\,(\lambda l\,.\\
\qquad\qquad\qquad\qquad\qquad (\mathsf{if0}\ (\mathsf{null?}\ l)\ \ulcorner 1 \urcorner\\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{if0}\ (\mathsf{car}\ l)\ (e\ \ulcorner 0 \urcorner)\\
\qquad\qquad\qquad\qquad\qquad\qquad (*\ (\mathsf{car}\ l)\ (\pi\ (\mathsf{cdr}\ l)))))))\\
\qquad\qquad\qquad\quad l)))
\end{array}
$$

Figure 1: The Function *product* in PCF with Lists

expression

$$(\textsc{Catch}\ e\ (\mathsf{add1}\ (e\ \ulcorner 0 \urcorner)))$$

returns $\ulcorner 0 \urcorner$ by popping the control stack back through the lexical binding of *e* in the form (CATCH *e* ...) after encountering the subexpression $(e\ \ulcorner 0 \urcorner)$. With such a CATCH construct, a procedure that multiplies the elements of a list of numbers can escape from the recursive traversal of the list if it determines that the list contains $\ulcorner 0 \urcorner$; see Figure 1 for the code. More importantly for our purposes, a program can use CATCH to determine the order of evaluation of a procedure's arguments. For example, the procedure

$$G = (\lambda f\,.\,(\textsc{Catch}\ x\ (f\ (x\ \ulcorner 0 \urcorner)\ (x\ \ulcorner 1 \urcorner))))$$

maps $+_l$ to $\ulcorner 0 \urcorner$ and $+_r$ to $\ulcorner 1 \urcorner$.

To equip PCF with non-local control, we add a family of procedures catch^τ of type $\tau \to o$ to PCF where $\tau = \tau_1 \to \ldots \tau_k \to o$.[7] If *f* is a procedure of type τ, then $(\mathsf{catch}^\tau\ f)$ returns $\ulcorner i-1 \urcorner$ if *f* evaluates the *i*th argument first and returns $\ulcorner k+n \urcorner$ if *f* is a constant procedure with result $\ulcorner n \urcorner$. Equivalently, $(\mathsf{catch}^\tau\ f)$ is $\ulcorner i-1 \urcorner$ if, for an appropriate list of variables, $y_1^{\sigma_1} \ldots y_m^{\sigma_m}$,

$$(f\ \underbrace{\Omega\ \ldots\ \Omega}_{i-1}\ \lambda y_1^{\sigma_1} \ldots y_m^{\sigma_m}.\mathsf{e}_j\ \underbrace{\Omega\ \ldots\ \Omega}_{k-i})$$

returns e_j for $j = 1, 2$. If $\tau = o$, catch^τ is degenerate: it always returns the value of its argument ($\ulcorner n \urcorner$).

In the context of PCF, the family of catch procedures has the same expressive power as Scheme's CATCH construct [7:39–40], which is equivalent to call/cc restricted to continuations of type $o \to \tau$ for some τ. We use the catch procedures in our extension of PCF because the catch procedures have a simpler definition and they facilitate proving the representability lemma (see Section 7). We designate our sequential extension of PCF by the name SPCF.

Definition 4.1 (*SPCF*) Let $\mathbb{E}$ be a set of error constants. The language SPCF($\mathbb{E}$) is PCF augmented by the set of constants $\mathbb{E} \cup \{\mathsf{catch}^\tau\}$:

$$
\begin{array}{lcll}
c & ::= & e & \quad (e \in \mathbb{E})\\
f & ::= & \mathsf{catch}^\tau. &
\end{array}
$$

[7]Indeed, we only need **catch** of type $(o \to \ldots \to o) \to o$ for the purpose of this paper.

The typing rules for the new constants are

$$A \vdash e : o \qquad \text{for all constants } e \in \mathbb{E}$$
$$A \vdash \mathsf{catch}^{\tau} : \tau \to o$$

SPCF is an abbreviation for SPCF($\mathbb{E}$) when the set $\mathbb{E}$ of error constants can be deduced from context.

Since the behavior of the procedure catch^{τ} depends only on the arity (see Definition 3.1) of the type τ, it suffices to decorate catch with the arity k instead of its complete type. We will frequently exchange the type superscript τ in favor of its arity k, using catch^{k} to denote any procedure catch^{τ} where τ has the form $(\tau_1 \to \ldots \tau_k \to o)$. ∎

4.3 Observably Sequential Programming Languages

Once we accept the idea that PCF should be extended to include error values and control operators like catch, we need to identify the technical properties that distinguish the extension from PCF. To this end, we introduce the notions of manifest sequentiality and observable sequentiality.

Definition 4.2 (*Manifestly Sequential and Observably Sequential Languages***)** A sequential PCF-like language L (see Definition 3.7) with semantics $\mathcal{C}$ is *manifestly sequential*[8] if there exists a non-empty set of error elements $\mathbb{E} \subseteq \mathcal{C}^o \setminus \{\bot\}$ denoted by a corresponding set of closed expressions $L_{\mathbb{E}}$ such that for any context $C[\;]_1 \ldots [\;]_k$ with sequentiality index j and any $E \in L_{\mathbb{E}}$,

$$\mathcal{C}[\![\, \vdash C[M'_1, \ldots, M'_{j-1}, E, M'_{j+1}, \ldots, M'_k]\,]\!] = \mathcal{C}[\![\, \vdash E\,]\!]\,.$$

A sequential PCF-like language L is *observably sequential* if for any context $C[\;]_1 \ldots [\;]_k$ with sequentiality index j, j is unique and there exists a program context $D[\;]$ such that

$$\mathcal{C}[\![\, \vdash D[\lambda x_1 \ldots x_k.C[x_1, \ldots, x_k]]\,]\!] = \mathcal{C}[\![\, \vdash \ulcorner j \urcorner\,]\!]\,. \;∎$$

SPCF is observably sequential and manifestly sequential (assuming $\mathbb{E}$ is non-empty). We will prove this fact after we define the semantics of SPCF in Section 7 (Theorem 7.5).

4.4 Procedure Denotations Have Explicit Computational Structure

Conventional semantic definitions for languages like SPCF rely on passing continuations, which are functions representing control contexts. In a continuation-passing semantics, the meaning of a phrase is a functions of it free variables *and* its control context. In essence, the function denoted by a phrase takes a "hidden" argument not manifest in the program

[8] If $\mathbb{E}$ contains at least two elements, a weaker property of languages called *error-sensitivity* is equivalent to manifest sequentiality. A PCF-like language L with semantics $\mathcal{C}$ is *error-sensitive* if there exists a non-empty set of error elements $\mathbb{E} \subseteq \mathcal{C}^o$ denoted by a corresponding set of closed expressions $L_{\mathbb{E}}$ such that if $\bot = \mathcal{C}[\![\, \vdash C[\Omega]\,]\!] \sqsubset \mathcal{C}[\![\, \vdash C[M]\,]\!]$ for any M then for all $E \in L_{\mathbb{E}}$, $\mathcal{C}[\![\, \vdash C[E]\,]\!] = \mathcal{C}[\![\, \vdash E\,]\!]$. In the presence of at least two errors, error-sensitivity implies sequentiality. We use the stronger notion of manifest sequentiality in this paper to address the case where there is only one element.

syntax, namely a continuation representing the control context. In a continuation semantics, control operators like CATCH are interpreted as functions that "grab" the continuation argument and pass it to other higher order arguments. For sequential languages, continuation-passing semantics are not fully abstract because they rely on continuous function spaces containing parallel functions (as well as control delimiters) [30]. Consequently, to construct a fully abstract semantics for SPCF, we must devise a more restrictive collection of domains that exclude parallel functions. The domains used in continuation semantics contain too many elements.

The informal operational semantics of SPCF, notably the behavior of the catch^k operators, suggests that procedure denotations should have more internal structure than function graphs. In particular, they should reflect the order in which arguments are evaluated. Consider the procedure

$$p \quad = \quad \lambda wxyz\,.\,(\mathsf{if0}\;x\;(y + \ulcorner 1 \urcorner)\;(z - \ulcorner 2 \urcorner)). \tag{1}$$

It has type $o \rightarrow (o \rightarrow (o \rightarrow (o \rightarrow o)))$, *i.e.*, it is a procedure mapping quadruples of integers to integers. The procedure p evaluates its second argument x first, but the subsequent evaluation order depends on the value of x. If x is 0, p evaluates its third argument y next; otherwise it evaluates its fourth argument z.

By supplying error values as arguments, a programmer can determine the schedule that p uses to evaluate its arguments. The results of these experiments can be expressed in the form of equations as follows:

$$\begin{array}{cccccccc}
(p & \Omega & \mathsf{e}_i & \Omega & \Omega) & = & \mathsf{e}_i \\
(p & \Omega & \ulcorner 0 \urcorner & \mathsf{e}_i & \Omega) & = & \mathsf{e}_i \\
(p & \Omega & \ulcorner n+1 \urcorner & \Omega & \mathsf{e}_i) & = & \mathsf{e}_i .
\end{array}$$

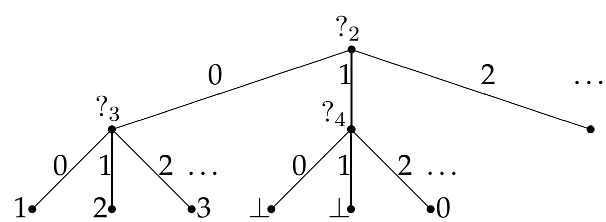
The equations demonstrate that p evaluates its arguments in a specific order and that the graph of the function denoted by p contains the schedule information.

The presence of error elements forces a function to follow a unique schedule. In the presence of errors, no function is strict in more than one piece of accessible, unexplored information. Consequently, we can represent functions by decision-trees (schedules) without sacrificing extensionality! For example, p can be represented as the index 2 paired with a "branching" function that maps the value of the second argument to appropriate information. The rest of the denotation is described in the same manner: if the second argument is 0, p maps the third argument y to $y + 1$; otherwise, it skips that argument and decreases the value of the fourth argument by 2:

$$
\langle 2, \left\{
\begin{array}{l}
\bot \mapsto \bot \\
\mathtt{e}_1 \mapsto \mathtt{e}_1 \\
\ldots \\
0 \mapsto \langle 3, \left\{
\begin{array}{l}
\bot \mapsto \bot \\
\mathtt{e}_1 \mapsto \mathtt{e}_1 \\
\ldots \\
0 \mapsto 1 \\
1 \mapsto 2 \\
\ldots
\end{array}
\right. \rangle \\
1 \mapsto \langle 4, \left\{
\begin{array}{l}
\bot \mapsto \bot \\
\mathtt{e}_1 \mapsto \mathtt{e}_1 \\
\ldots \\
0 \mapsto \bot \\
1 \mapsto \bot \\
2 \mapsto 0 \\
3 \mapsto 1 \\
\ldots
\end{array}
\right. \rangle \\
\ldots
\end{array}
\right. \rangle
$$

Since p must be manifestly sequential, its branching functions map $\bot$ to $\bot$ and $\mathtt{e}_i$ to $\mathtt{e}_i$.

Rotating the preceding picture produces a tree with decision points as internal nodes and alternative outcomes as branches:



For readability, each argument index i has been replaced by the symbol $?_i$ in the rotated picture. A node with the label $?_i$ represents the querying of the ith argument. This notation emphasizes that the respective indices are queries about an argument; it will be generalized in the next subsection to a representation where procedures can play the role of arguments. The edges below a node with label $?_i$ are labeled with the possible values of the ith argument. The target node for each edge specifies what action the procedure performs next. If it is a leaf (an element of $\mathbb{N} \cup \{\bot, \mathtt{e}_1, \ldots\}$), then p has completed its computation; otherwise, it continues to query its argument(s).

The most important property of these trees is that they are extensional when the set of errors is non-empty. Every manifestly sequential function has a *unique* decision-tree representation.

4.5 Higher-order Procedures Explore and Output Trees Sequentially

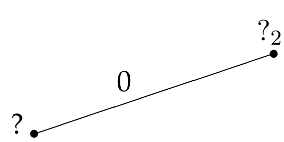
Given that we want procedures to denote decision trees, our next task is to explain how higher-order procedures gather information about procedural inputs. Since trees resemble LISP S-expressions, we interpret higher-order procedures as processes that examine decision trees in essentially the same fashion as LISP procedures traverse S-expressions. Like a LISP procedure, a higher-order procedure can inspect a node in an input tree only after inspecting its predecessors. We express queries about procedural inputs as patterns that identify nodes in decision trees. Thus, a query is a pair consisting of a *path-pattern* and an argument index. A path-pattern is a rooted finite path of nodes and edges in a decision tree terminated by the marker "?". The marker identifies the node that the procedure wants to inspect.[9] If g is the actual argument under inspection, the *response* to a query is the label of the specified node in g.

The crucial semantic restriction on the process is that it is sequential and propagates error values: it cannot ignore infinite loops or errors in inputs. If computing an output node requires inspecting an input node that is $\perp$ or e_i, the computed output node must be $\perp$ or e_i, respectively.

To illustrate what happens when a function explores a higher order input (a tree), let us examine the behavior of the procedure

$$F = \lambda g.(\mathtt{add1}\ (g\ \Omega\ \ulcorner 0 \urcorner\ \ulcorner 2 \urcorner\ \mathsf{e}_1)) : (o \to o \to o \to o \to o) \to o.$$

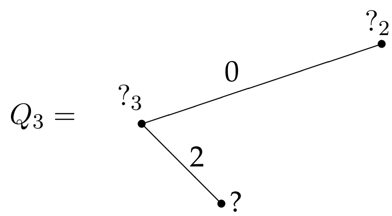
Since this procedure has only one argument, all the queries in the denotation of F have the form $\langle Q, 1\rangle$. When (the denotation of) F is applied to an argument tree g, it knows nothing about the value of g. Since F must determine how g behaves on certain arguments, F's initial query is $Q_1 = ?$ to determine the root of g. If the value of g is the procedure p described in the preceding subsection, g answers this question with the response $?_2$, which is just the label of the root of p. Now F knows that g initially demands information about its second argument. To find out more about g, the procedure F must provide the information that g has requested, which is 0. This information determines the subtree within g that F wants to inspect. So F asks the question $\langle Q_2, 1\rangle$ where

$Q_2 =$ 

As with Q_1, Q_2 is a pattern identifying the node of g that F wants to inspect; it lies just below the root of g on the edge labeled 0. Matching the pattern Q_2 against p yields the label $?_3$, which is a query about p's third argument. Thus, as long as the input demands more information about its arguments, its responses will simply answer the queries (about g) with queries (about the arguments of g).

The procedure F passes the value 2 as the third argument to g. Consequently, F's next question to g is $\langle Q_3, 1\rangle$ where:

[9] According to this convention, the queries $?_1, ?_2, \ldots$ should be written as $\langle 1, ?\rangle, \langle 2, ?\rangle, \ldots$, respectively. To make our examples more readable, we elected to use the more compact notation $?_i$ instead of $\langle i, ?\rangle$.



Matching this pattern against p yields the leaf value 3, which is the final answer of p along this path. This response from p completes the subcomputation $(g\ \Omega\ 0\ 2\ \mathsf{e}_1)$. F subsequently adds 1 to the result of the subcomputation, yielding the final answer 4.

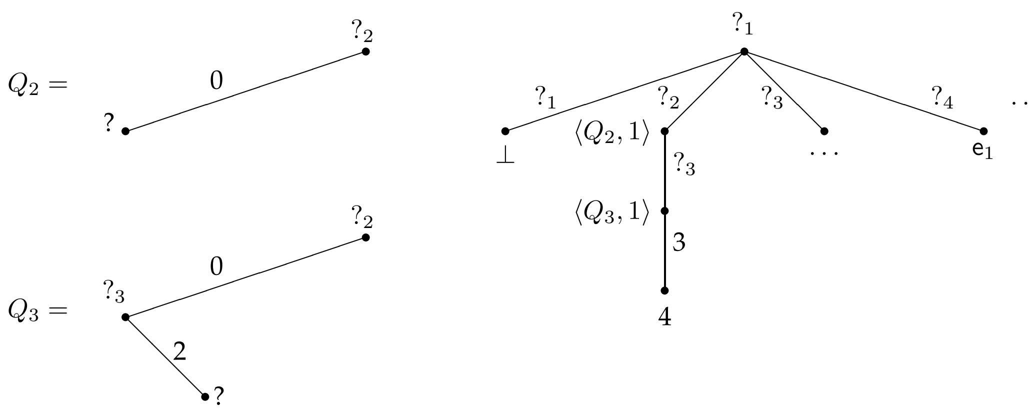


Figure 2: A Fragment of Procedure *F*

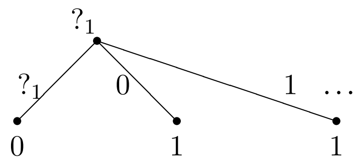
If *F* were a black box, a programmer could determine the schedule for *F* by applying *F* to error-laden arguments. A portion of the schedule for F (which is infinite), including the cases discussed above, is captured by the following equations:

$$
\begin{array}{lll}
(F\quad (\lambda wxyz\,.\,\mathsf{e}_1)) & = & \mathsf{e}_1 \\
(F\quad (\lambda wxyz\,.\,\mathsf{e}_2)) & = & \mathsf{e}_2 \\
(F\quad (\lambda wxyz\,.\,(\mathsf{if0}\ x\ \mathsf{e}_1\ \mathsf{e}_2)) & = & \mathsf{e}_1 \\
(F\quad (\lambda wxyz\,.\,(\mathsf{if0}\ x\ (\mathsf{if0}\ (\mathsf{sub1}\ (\mathsf{sub1}\ y))\ \mathsf{e}_1\ \mathsf{e}_2)\ \mathsf{e}_3)) & = & \mathsf{e}_1 \\
(F\quad (\lambda wxyz\,.\,(\mathsf{if0}\ x\ (\mathsf{if0}\ (\mathsf{sub1}\ (\mathsf{sub1}\ y))\ 3\ \mathsf{e}_2)\ \mathsf{e}_3)) & = & 4 \\
 & & \\
(F\quad (\lambda wxyz\,.\,(\mathsf{if0}\ z\ \mathsf{e}_1\ \mathsf{e}_2))) & = & \mathsf{e}_1 \\
(F\quad (\lambda wxyz\,.\,(\mathsf{if0}\ z\ \mathsf{e}_2\ \mathsf{e}_3))) & = & \mathsf{e}_1
\end{array}
$$

The last pair of experiments shows that *F* produces the constant output e_1 for a procedure argument that is strict in its fourth position. It also shows why it is necessary in some cases to repeat an experiment using different errors.

In a domain of decision trees representing schedules, we must be able to represent the schedule for F by a tree with queries to inputs as internal nodes, final answers as leaves, and possible answers from inputs as edges. Figure 2 shows the portion of tree denoted by *F*, captured in the equations above. It also includes a leaf showing that *F* diverges if its procedure argument g immediately demands its fourth argument. The other branches in the tree representing *F* have been omitted.

Finally, when procedures are represented as trees, it is easy to see that higher-order procedures can extract apparently intensional information from procedural arguments. For example, the following tree represents a procedure that maps constant unary procedures to 1 and strict unary procedures to 0:



As we will see in Section 6 below, the `catch` procedure denotes a similar tree or manifestly sequential function.

4.6 SPCF Defines Manifestly Sequential Functions

In summary, the analysis of PCF from the perspective of language design shows that adding control operators to PCF produces a more general form of computation in which functions have a natural representation as decision trees. Moreover, if the domain of computation includes error elements, the decision tree representation is extensional. The remainder of the paper is devoted to constructing a fully abstract semantics for the family of languages SPCF($\mathbb{E}$), which extend PCF by the control operators catch$^{\tau}$ and the error elements $\mathbb{E}$.

This work expands and generalizes the work by Berry and Curien [4] on semantics for sequential languages based on *sequential algorithms* over *concrete data structures*. The next section presents a new framework derived from concrete data structures—called *sequential data structures*—suitable for defining both Berry and Curien's sequential algorithms and our manifestly sequential functions. This construction yields a cartesian-closed category of manifestly sequential domains and manifestly sequential functions, which forms the basis for defining fully abstract semantics for the languages SPCF($\mathbb{E}$).

5 Sequential Data Structures with Errors

The definition of a suitable semantics for SPCF presumes the existence of a category of domains where the arrows are functions represented by decision-trees as described in the preceding section. In denotational semantics, domains are usually constructed as the ideal completions of finitary bases. The completion process adds infinite elements as limit points for directed sets of finite elements[27]. Our construction of domains of trees follows the same general pattern, but relies on smaller bases consisting exclusively of prime elements.[10] This construction is less general than the usual one, but it yields a simpler representation for domain elements in terms of the original basis.

To formulate decision trees as a Scott domain, we need to identify the finite pieces of information used to build decision trees. For reasons that will become clear later, we have elected to represent trees as sets of paths. Each path consists of an alternating sequence of

[10] Kanneganti, Cartwright, and Felleisen [19] developed an alternate characterization of the same categorical framework using a topological formulation of the function space construction.

addresses and data items. Every path begins with an address.[11] Each address on a path identifies which son below the current tree node is being selected. A tree is a set of paths ending in data items that satisfies two constraints. First, the set is closed under the prefix ordering on paths ending in data items. Second, all common prefixes between paths end in data items. Hence, the first point of disagreement between any two incomparable paths is an address.

A framework of addresses, data items, and paths governing the construction of trees is called a *sequential data structure*. We will subsequently show that the set of trees determined by a sequential data structure form a Scott domain.

Definition 5.1 (*Sequential Data Structure, Path, Query, Response***)** A sequential data structure (*sds*) **M** is a 3-tuple (A, D, P) consisting of three countable sets:

- a set A of *addresses*;
- a set D of *data*; and
- a prefix-closed set P of non-empty, *alternating paths* in (A,D)*, *i.e.*,
  1. $P \subseteq (A, D)^*$ (see Section 2 (Paths));
  2. $\epsilon \notin P$;
  3. (prefix-closure) if $p \in P$, $p' \in (A, D)^*$, $p' \neq \epsilon$, and $p' \sqsubseteq p$ (where $\sqsubseteq$ is the prefix-ordering on paths) implies $p' \in P$.

A path ending in an address is also called *query*, and a path ending in a datum is called *response*; $Que_{\mathbf{M}}$ and $Res_{\mathbf{M}}$ denote the sets of queries and responses for a sequential data structure **M**, respectively. ∎

A simple example that illustrates the concept of an *sds* is the following structure, which generates the natural numbers as "trees":

$$\mathbf{N} = (\{?\}, \mathbb{N}, \{?,\ ? \cdot n \mid n \in \mathbb{N}\}).$$

N uses a single address, ?. At this address a path of **N** contains a single number, if it contains any datum at all. Similarly, an *sds* **T** for truth values has the following definition:

$$\mathbf{T} = (\{?\},\ \{tt, \mathit{ff}\},\ \{?, ? \cdot tt, ? \cdot \mathit{ff}\}).$$

A slightly more interesting example is the *sds* $\mathbf{N} \times \mathbf{T}$, which represents the *cartesian* product of **N** and **T**:

$$\mathbf{N} \times \mathbf{T} = (\{?_1, ?_2\}, \mathbb{N} \cup \{tt, \mathit{ff}\}, \{?_1, ?_2, ?_1 \cdot n, ?_2 \cdot tt, ?_2 \cdot \mathit{ff} \mid n \in \mathbb{N}\}).$$

Depending on the first address, a path in $\mathbf{N} \times \mathbf{T}$ indicates the presence of a truth value *or* the presence of a number in a pair. Since a path contains exactly one address and since

[11] To construct a cartesian-closed category, we must construct a collection of objects that is closed under cartesian products. Hence, we need to construct a domain of tuples of decision trees, which are forests. *Par abus de langage* we continue to speak of trees.

one address is associated with truth values and the other with numbers, a path never represents a complete pair but one of the two components. In contrast, the following *sds* $\mathbf{N} \otimes \mathbf{T}$ is the basis for the *strict* product of the two *sds*'s:

$$\mathbf{N} \otimes \mathbf{T} = (\{?\}, \mathbb{N} \times \{tt, \mathit{ff}\}, \{?, ? \cdot (n, tt), ? \cdot (n, \mathit{ff}) \,|\, n \in \mathbb{N}\}).$$

Here a path has an address for a pair of data components, namely, a truth value *and* a number.

As a last example, consider the *sds* $\mathbf{N}^4 \rightarrow \mathbf{N}$ whose trees are the denotations of procedures of type $o \rightarrow o \rightarrow o \rightarrow o \rightarrow o$ like the procedure *p* defined in equation (1). In general such a procedure may just output a result or it may inspect its input. In the first case, the procedure's representation is basically a single datum ($n \in \mathbb{N}$) but, to formulate this description as an *sds*-path, it is prefixed with an address ($?$). Intuitively, a path of the shape $? \cdot n$ represents a procedure that "fills" the only output address ($?$) with a natural number (n). In the second case, the procedure's paths must include a strategy for exploring the arguments. A straightforward representation of a strategy is a series of query-response pairs, where the query indicates which part of the argument is inspected and the response indicates the anticipated response. In the *sds* $\mathbf{N}^4 \rightarrow \mathbf{N}$, the argument is a four-tuple of numbers and hence provides four possible choices for inspection, all of which have numbers as the only possible responses. The symbols $?_1, ?_2, ?_3$, and $?_4$ denote queries and numbers specify responses.

The representation of a non-constant procedure is a set of paths that all start with the address $?$. Each path describes a strategy for exploring the input addresses $?_1, ?_2, ?_3$, and $?_4$, which are *data items* in the representation of the procedure. The path returns an "answer" in $\mathbb{N}$ when it has gathered sufficient information about the contents of the input addresses. The paths are identical to the paths in the tree representation for p sketched in Subsection 4.4, except that they are preceded by the address $?$. Thus, some paths in the denotation of the procedure *p* are $?{\cdot}?_2 \cdot 0{\cdot}?_3 \cdot 1 \cdot 2$ and $?{\cdot}?_2 \cdot 1{\cdot}?_4 \cdot 2 \cdot 0$.

The formal specification of the *sds* $N^4 \rightarrow N = (A_\rightarrow, D_\rightarrow, P_\rightarrow)$ for representations of procedures like *p* is as follows:

$$\begin{aligned} A_\rightarrow &= \{?\} \cup \mathbb{N} \\ D_\rightarrow &= \{?_1, ?_2, ?_3, ?_4\} \cup \mathbb{N} \end{aligned}$$

$$P_{\rightarrow} \;=\; \left\{ \begin{array}{l|l} ?, & \\ ?\cdot n, & \\ ?\cdot ?_{i_1}, & \\ ?\cdot ?_{i_1}\cdot n_1, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot n, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2, & n, n_1, n_2, n_3, n_4 \in \mathbb{N}, \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot n, & \{i_1, i_2, i_3, i_4\} = \{1,2,3,4\} \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot ?_{i_3}, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot ?_{i_3}\cdot n_3, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot ?_{i_3}\cdot n_3 \cdot n, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot ?_{i_3}\cdot n_3 \cdot ?_{i_4}, & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot ?_{i_3}\cdot n_3 \cdot ?_{i_4}\cdot n_4 & \\ ?\cdot ?_{i_1}\cdot n_1 \cdot ?_{i_2}\cdot n_2 \cdot ?_{i_3}\cdot n_3 \cdot ?_{i_4}\cdot n_4 \cdot n & \end{array} \right\}$$

Equivalently, a path is a member of the context-free language (specified in extended BNF):

$$\begin{array}{lcl} p & ::= & r \mid q \\ r & ::= & ?\cdot[?_i \cdot n]^* \cdot m \mid ?\cdot[?_i \cdot n]^* \cdot ?_j \\ q & ::= & ?\cdot[?_i \cdot n]^* \end{array}$$

where $m, n \in \mathbb{N}$, $i, j \in \{1,2,3,4\}$ and no address $(?, ?_i)$ occurs more than once in a path.

The formal definition of $\mathbf{N}^4 \rightarrow \mathbf{N}$ also clarifies how a query of the *sds* $\mathbf{N}$ becomes a datum in the specification of a procedure space, and, conversely, how a datum in $\mathbf{N}$ becomes an address in the procedure space.

Given the preceding definition of *sds*'s, we can restate our informal definition of a tree as follows. A tree is a set of responses such that all the response predecessors of a response are included and such that all non-empty intersections of elements are responses (and hence included). In the domain of trees determined by an *sds*, the finite elements are precisely the finite trees, including the empty one.

To accommodate error values in computations involving trees, we need to include error elements in the domain of trees determined by an *sds*. The following definition formalizes the definition of trees and tree domains that possibly contain error values.

Definition 5.2 ($($*Manifestly*$)$ *Sequential Domain, Tree***)** Let $\mathbf{M} = (A, D, P)$ be an *sds*, and let

$$\mathbb{E} = \{\mathsf{e}_1, \mathsf{e}_2, \ldots\}$$

be a possibly empty set of error values, or errors for short, disjoint from D. An *error response* is a path $r = q \cdot \mathsf{e}$ consisting of a query $q \in Que_{\mathbf{M}}$ extended with an element $\mathsf{e} \in \mathbb{E}$. Since $\mathbb{E} \cap D = \emptyset$, error responses and responses are disjoint. An *observable response r* is either a response or an error response:

$$Res^{e}_{\mathbf{M}} = Res_{\mathbf{M}} \cup \{q : \mathsf{e} \mid q \in Que_{\mathbf{M}}, \mathsf{e} \in \mathbb{E}\}.$$

A *tree x* over $\mathbf{M}$ relative to $\mathbb{E}$ is a set of observable responses closed under the prefix ordering and greatest lower bounds:

prefix-closed: $x \subseteq Res^e_{\mathbf{M}}$ such that if $r \in x$ and $r' \sqsubseteq r$ for some $r' \in Res_{\mathbf{M}}$, then $r' \in x$;

glb-closed: if $r, r' \in x$, then $r \sqcap r' = \epsilon$ or $r \sqcap r' \in Res^e_{\mathbf{M}}$ (and, by prefix-closure, is therefore in *x*).

A tree that does not contain error responses is called *error-free*. The set of trees over $\mathbf{M}$ relative to $\mathbb{E}$ is denoted by $\mathbb{D}(\mathbf{M})$, and is ordered by set-inclusion. A partial order that is isomorphic to $\mathbb{D}(\mathbf{M})$ for some $\mathbf{M}$ is called the *sequential domain generated by* $\mathbf{M}$. $\mathbb{D}_0(\mathbf{M})$ is the set of finite sets of paths (finite trees). We call these domains *manifestly sequential* if the underlying set of errors is non-empty. ∎

Convention The symbol $\perp$ is used interchangeably with $\emptyset$ for the least element in domains.

To provide some intuition about sequential domains, let us describe the domains generated by the example *sds*'s presented earlier in this section. Unless noted otherwise, this discussion of examples assumes that the set $\mathbb{E}$ of errors is empty.

The domain $\mathbb{D}(\mathbf{N})$ generated by $\mathbf{N}$ has as its universe the set $\{\perp, \{? \cdot n\} \,|\, n \in \mathbb{N}\}$. It is clearly isomorphic to the flat domain $\mathbb{N}_\perp$ of natural numbers. If the set of errors is non-empty, $\mathbb{D}(\mathbf{N})$ contains all the error elements, which are incomparable to numbers and each other but dominate bottom. The composite symbol $\mathbb{N}^e_\perp$ denotes the domain generated by $\mathbf{N}$ when errors are present.

The domain $\mathbb{D}(\mathbf{T})$ is flat like $\mathbb{D}(\mathbf{N})$; it is isomorphic to $\mathbb{T}_\perp$, the flat domain of truth values. Again, if there are errors, each of them dominates $\perp$ and is incomparable to all other elements.

The domain $\mathbb{D}(\mathbf{N} \times \mathbf{T})$ consists of the set

$$\{\perp, \{?_1 \cdot n\}, \{?_2 \cdot t\}, \{?_1 \cdot n, ?_2 \cdot t\} \,|\, n \in \mathbb{N}, t \in \{tt, \mathit{ff}\}\}$$

under the relation $\subseteq$. It is isomorphic to the cartesian product domain $\mathbb{N}_\perp \times \mathbb{T}_\perp$. This example illustrates how independent information appears on distinct paths in a tree and how a missing path in a tree signifies lack of information (bottom). Consider the trees $\{?_1 \cdot 1, ?_2 \cdot tt\}$ and $\{?_2 \cdot tt\}$; the former represents the pair $(1, tt)$ and the latter corresponds to the pair $(\perp, tt)$.

The domain $\mathbb{D}(\mathbf{N} \otimes \mathbf{T})$ over the *sds* $\mathbf{N} \otimes \mathbf{T}$ is clearly isomorphic to the strict product domain $\mathbb{N}_\perp \otimes \mathbb{T}_\perp$. The elements of $\mathbb{D}(\mathbf{N} \otimes \mathbf{T})$ are $\{\perp, \{? \cdot (n, t)\} \,|\, n \in \mathbb{N}, t \in \{tt, \mathit{ff}\}\}$. In other words, every element of $\mathbb{D}(\mathbf{N} \otimes \mathbf{T})$ other than $\perp$ is the single query $?$ followed by a pair (n, t) where $n \in \mathbb{N}$ and and $t \in \{\mathsf{tt}, \mathsf{ff}\}$.

In the absence of errors, the domain $\mathbb{D}(\mathbf{N}^4 \to \mathbf{N})$ does *not* correspond to a standard function domain construction. It is isomorphic to Berry and Curien's domain of sequential algorithms [8:ch. 2]. As we have already pointed out, sets of paths in $\mathbf{N}^4 \to \mathbf{N}$ are most easily understood as trees (technically forests). Unlike the trees of the preceding examples whose maximal depth was two, the elements of $\mathbb{D}(\mathbf{N}^4 \to \mathbf{N})$ may be up to ten levels deep. Two examples of error-free trees over this structure are

$$\begin{aligned} \Pi^1_l &= \{?{\cdot}?_1, ?{\cdot}?_1 \cdot n \cdot n \,|\, n \in \mathbb{N}\} \\ \Pi^1_s &= \{?{\cdot}?_1\} \end{aligned}$$

Both "algorithms" attempt to provide a datum for their single output address ? and, for this purpose, inspect the first component of their argument. The first algorithm returns the response it gets for its query and thus computes the projection function of four-tuples for the first position. The second one is an approximation to the first: it diverges after the inspection of the argument yields an error-free response. These examples serve to illustrate some concepts in the following discussion. While $\mathbb{D}(\mathbf{N}^4 \to \mathbf{N})$ is not isomorphic to a traditional domain construction, it is nevertheless a Scott domain as are all domains over an *sds*.

Proposition 5.3 *The (observably) sequential domain* $\mathbb{D}(\mathbf{M})$ *over an sds* $\mathbf{M}$ *is a Scott domain; the corresponding set of finite trees* $\mathbb{D}_0(\mathbf{M})$ *comprises the finite elements of* $\mathbb{D}(\mathbf{M})$.

Proof. It is easy to verify that the union of a bounded set of trees is a tree, that is, that the union is also prefix- and glb-closed. Hence, the proof of the proposition is a variation of the proof for the standard meta-theorem on ideal constructions [27]. ∎

After establishing what the domains for the semantics of SPCF are, we turn to functions between domains. Specifically, we must formalize the notion of a manifestly sequential function, which we sketched in the previous section. We begin by introducing some auxiliary terminology and notation.

Definition 5.4 (*Open and Answered Query***)** Let $\mathbf{M} = (A, D, P)$ be an *sds*. Given a tree x in the domain $\mathbb{D}(\mathbf{M})$, a query $q \in Que_{\mathbf{M}}$ is:

- *answered* in x, written $q \in \mathit{Answered}(x)$, if $q \sqsubseteq r$ for some $r \in x$;
- *open* in x, written $q \in \mathit{Open}(x)$, if q is not answered in x and $q = r \cdot a$, for some $r \in x$ and $a \in A$.

∎

To illustrate this terminology, consider the domain elements Π^1_l, Π^1_s in $\mathbb{D}(\mathbf{N}^4 \to \mathbf{N})$:

$$\begin{array}{lcllcl} \mathit{Open}(\Pi^1_l) & = & \emptyset & \mathit{Answered}(\Pi^1_l) & = & \{?, ?{\cdot}?_1 \cdot n \mid n \in \mathbb{N}\} \\ \mathit{Open}(\Pi^1_s) & = & \{?{\cdot}?_1 \cdot n \mid n \in \mathbb{N}\} & \mathit{Answered}(\Pi^1_s) & = & \{?\}. \end{array}$$

Next we turn to the definition of the sequential and manifestly sequential functions between domains determined by *sds*'s. Our definitions of these terms are motivated by constraints on the meaning of contexts imposed by sequential and manifestly sequential languages.

Definition 5.5 (*Sequential and Manifestly Sequential Functions***)** Let $\mathbf{M}_1$, $\mathbf{M}_2$ be *sds*'s and let $\mathbb{D}(\mathbf{M}_1), \mathbb{D}(\mathbf{M}_2)$ be the corresponding domains relative to an error set $\mathbb{E}$. A function $f : \mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ is:

- *monotonic* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$;
- *continuous* if $r \in f(x)$ implies $r \in f(y)$ for some finite $y \sqsubseteq x$.[12]

[12] This notion of continuity is weaker than the usual definition based on directed sets, but in tandem with monotonicity it is equivalent to the usual definition (which implies monotonicity).

A monotonic, continuous function $f : \mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ is:

- *sequential* if $q' \in \mathit{Open}(f(x))$ and $q' \in \mathit{Answered}(f(z))$ for some finite $z \sqsupseteq x$ implies that there exists $q \in \mathit{Open}(x)$, called a *sequentiality index* of f at (x, q'), such that for all $y \sqsupseteq x$, $q' \in \mathit{Answered}(f(y))$ implies $q \in \mathit{Answered}(y)$;

- *manifestly sequential* if f is sequential, $\mathbb{E}$ is non-empty, and for any sequentiality index q at (x, q'), $f(x \cup \{q \cdot \mathsf{e}\}) = f(x) \cup \{q' \cdot \mathsf{e}\}$ for $\mathsf{e} \in \mathbb{E}$;[13]

For $\mathbb{E} \neq \emptyset$, we let $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ denote the collection of manifestly sequential functions from $\mathbb{D}(\mathbf{M}_1)$ to $\mathbb{D}(\mathbf{M}_2)$. If $\mathbb{E} = \emptyset$, then no function $f : \mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ is manifestly sequential because manifest sequentiality requires $\mathbb{E} \neq \emptyset$. In this case we let $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ denote the collection of sequential functions from $\mathbb{D}(\mathbf{M}_1)$ to $\mathbb{D}(\mathbf{M}_2)$. ∎

To explain the concept of manifest sequentiality and its relationship to sequentiality, let us examine a series of examples. First, we observe that a continuous, monotonic function may not have a sequentiality index for a given pair of input value and output query. Let $f : \mathbb{D}(\mathbf{N} \times \mathbf{T}) \longrightarrow \mathbb{D}(\mathbf{T})$ be defined as follows:

$$\begin{array}{rcl} f(\{?_1 \cdot 0\} \cup X) & = & \{? \cdot tt\} \\ f(\{?_2 \cdot tt\} \cup X) & = & \{? \cdot tt\} \\ f(\{?_1 \cdot m, ?_2 \cdot \mathit{ff}\}) & = & \{? \cdot \mathit{ff}\} \text{ for } m \geq 1 \\ f(\bot) & = & \bot \end{array}$$

where $X \supseteq \emptyset$ is any set such that the input is a proper tree. The function is not sequential because f does not have a sequentiality index at $(\bot, ?)$: f can produce output by looking at either $?_1$ or $?_2$ and thus neither of them is a sequentiality index. A uniprocessor implementation of non-sequential function must rely on time-slicing to prevent a diverging input from forcing the implementation to diverge.

The identity function $\mathsf{id} : \mathbb{D}(\mathbf{N}) \longrightarrow \mathbb{D}(\mathbf{N})$ (relative to $\mathbb{E} = \{\mathsf{e}\}$) is manifestly sequential. The query $?$ is the sequentiality index for $(\bot, ?)$. No other pair can have a sequentiality index, and thus id is sequential. The sequentiality index of the identity function from $\mathbb{D}(\mathbf{N}^4 \longrightarrow \mathbf{N})$ to itself (over the empty set of errors) for $(\bot, ?)$ is $?$. For $(x, ?{\cdot}?_1 \cdot n)$ (with $?{\cdot}?_1 \cdot n \in \mathit{Open}(x)$), it is $?{\cdot}?_1 \cdot n$. Again, the function is sequential. Indeed, all identity functions are manifestly sequential functions. Moreover, the composition of two manifestly sequential functions is manifestly sequential.

Proposition 5.6 *Let $\mathbf{M}$, $\mathbf{M}_1$, $\mathbf{M}_2$ and $\mathbf{M}_3$ be sds's and let $\mathbb{D}(\mathbf{M})$, $\mathbb{D}(\mathbf{M}_1)$, $\mathbb{D}(\mathbf{M}_2)$, and $\mathbb{D}(\mathbf{M}_3)$ be the corresponding domains relative to $\mathbb{E}$. Then*

[13] If $\mathbb{E}$ contains at least two elements, then the weaker notion of *error-sensitivity* is equivalent to *manifest sequentiality*. Given a monotonic, continuous function $f : \mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$, the input $z \in \mathbb{D}(\mathbf{M_1})$ is a *threshold* for query q' if $q' \in \mathit{Answered}(f(z))$ yet $q' \in \mathit{Open}(f(y))$ for all $y \sqsubset z$. For $\mathsf{e}_i \in \mathbb{E}$, an e_i*-variant* of $z \in \mathbb{D}(\mathbf{M_1})$ is an element of the form $y \cup \{q \cdot \mathsf{e}_i\}$ such that $y \sqsubseteq z$, $\mathit{Open}(y) \subseteq \mathit{Open}(z) \cup \{q\}$. The function f is *error-sensitive* if for all finite $x, y, z \in \mathbb{D}(\mathbf{M_1})$ such that $x \sqsubset z \in \mathbb{D}(\mathbf{M_1})$, z is threshold for q', and y is an e_i-variant of z:

$$q' : \mathsf{e}_i \in f(y).$$

1. *the identity function* $\mathsf{id}^{\mathbb{D}(\mathbf{M})} : \mathbb{D}(\mathbf{M}) \longrightarrow \mathbb{D}(\mathbf{M})$ *is (manifestly) sequential: sequential for all* $\mathbb{E}$ *and manifestly sequential for* $\mathbb{E} \neq \emptyset$*;*

2. *if* $f : \mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ *and* $g : \mathbb{D}(\mathbf{M}_2) \longrightarrow \mathbb{D}(\mathbf{M}_3)$ *are (manifestly) sequential functions, then the composite function* $h(x) = g(f(x))$ *is (manifestly) sequential.*

Proof. The first part of the proposition follows from a generalization of the argument just given for the id function on the domain $\mathbb{D}(\mathbf{N}^4 \longrightarrow \mathbf{N})$. To verify the second part, we must prove that $g \circ f$ is monotonic, continuous, and (manifestly) sequential. The first two properties are easy to verify. To prove sequentiality, assume $q' \in \mathit{Open}(g(f(x)))$ and $q' \in \mathit{Answered}(g(f(z)))$ for some $z \sqsupseteq x$. Then, by the (manifest) sequentiality of g, there is a (unique) sequentiality index q^* for $(f(x), q')$. Clearly, $q^* \in \mathit{Open}(f(x))$ and $q^* \in \mathit{Answered}(f(z))$. Hence, by the (manifest) sequentiality of f, there is a (unique) sequentiality index q for (x, q^*). It is easy to check that q is a (unique) sequentiality index of $g \circ f$ for (x, q'). Finally, to verify manifest sequentiality, observe that

$$\begin{array}{rcll} g(f(x \cup \{q \cdot \mathsf{e}\})) & = & g(f(x) \cup \{q^* \cdot \mathsf{e}\}) & \text{by the manifest sequentiality of } f \\ & = & g(f(x)) \cup \{q' \cdot \mathsf{e}\} & \text{by manifest sequentiality of } g \end{array}$$

which concludes the proof. ∎

A more interesting example of a sequential function is the addition function from $\mathbb{D}(\mathbf{N} \times \mathbf{N})$ to $\mathbb{D}(\mathbf{N})$ relative to the empty error set:

$$+ : \left\{ \begin{array}{lcl} \mathbb{D}(\mathbf{N} \times \mathbf{N}) & \longrightarrow & \mathbb{D}(\mathbf{N}) \text{ for } \mathbb{E} = \emptyset \\ \{?_1 \cdot n_1, ?_2 \cdot n_2\} & \mapsto & \{? \cdot n \,|\, n = n_1 + n_2\} \\ \{?_1 \cdot n\} & \mapsto & \bot \\ \{?_2 \cdot n\} & \mapsto & \bot \\ \bot & \mapsto & \bot \end{array} \right.$$

for $n \in \mathbb{N}$. At the input $\bot$ and the output query $?$, the function has *two* sequentiality indices, namely, $?_1$ and $?_2$. Only if both addresses of the input tree contain a datum will $+$ produce a response distinct from $\bot$.

In contrast, a manifestly sequential function f has unique sequentiality indices and a non-empty error set $\mathbb{E}$.

Proposition 5.7 *Let* $\mathbf{M}_1$*,* $\mathbf{M}_2$ *be sds's and let* $\mathbb{D}(\mathbf{M}_1)$ *and* $\mathbb{D}(\mathbf{M}_2)$ *be the respective domains relative to* $\mathbb{E}$*. Let* $f : \mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ *be an manifestly sequential function. Then all sequentiality indices are unique.*

Proof. Let $x \in \mathbb{D}(\mathbf{M}_1)$ be a tree such that q' is open for $f(x)$ and answered in $f(z)$ for some $z \sqsupseteq x$. Assume q_1 and q_2 are sequentiality indices. Then, by manifest sequentiality,

$$f(x \cup \{q_1 : \mathsf{e}\}) = f(x) \cup \{q' : \mathsf{e}\}$$

for $\mathsf{e} \in \mathbb{E}$. Since q' is answered in $f(x) \cup \{q' : \mathsf{e}\}$ and $x \cup \{q_1 : \mathsf{e}\} \sqsupseteq x$, sequentiality implies that q_2 is answered in $x \cup \{q_1 : \mathsf{e}\}$. But given that q_2 was open in x, this means that $q_1 = q_2$, as desired. ∎

Notation: Based on this lemma, it makes sense to introduce the notation $si_f(x, q')$ for *the* sequentiality index of an observably sequential function f at (x, q'). ∎

As a result, the evaluation strategy for f is uniquely determined by the graph of the function. Moreover, that strategy can be effectively extracted from the graph by inspecting the behavior of f on inputs containing errors. There are two manifestly sequential addition functions. Left addition is defined by the graph:

$$+_l : \begin{cases} \mathbb{D}(\mathbf{N} \times \mathbf{N}) & \longrightarrow & \mathbb{D}(\mathbf{N}) \text{ for } \mathbb{E} \neq \emptyset \\ \{?_1 \cdot n_1, ?_2 \cdot n_2\} & \mapsto & \{? \cdot n \,|\, n = n_1 + n_2\} \\ \{?_1 \cdot n\} & \mapsto & \perp \\ \{?_1 \cdot \mathsf{e}_i\} & \mapsto & \{? \cdot \mathsf{e}_i\} \\ \{?_1 \cdot \mathsf{e}_i, ?_2 \cdot x\} & \mapsto & \{? \cdot \mathsf{e}_i\} \\ \{?_2 \cdot x\} & \mapsto & \perp \\ \{?_1 \cdot n, ?_2 \cdot \mathsf{e}_i\} & \mapsto & \{? \cdot \mathsf{e}_i\} \\ \perp & \mapsto & \perp \end{cases}$$

where $n \in \mathbb{N}$, $\mathsf{e}_i \in \mathbb{E}$, and $x \in \mathbb{N} \cup \mathbb{E}$. When $+_l$ inspects its input it always inspects the first component of the pair first. If the first component is e_i, manifest sequentiality forces $+_l$ to return e_i. If the first component is a number, then $+_l$ inspects the second component and returns the appropriate result when the value of the second component is determined. The reader may want to construct the graph of "right addition", $+_r$, and compare it with the graph of $+_l$.

The preceding definitions of manifestly sequential domains and functions and Proposition 5.6 have laid the groundwork for constructing a category $\mathcal{SEQ}(\mathbb{E})$ suitable for defining the semantics of of SPCF.

6 The Manifestly Sequential Cartesian-Closed Category

In this section we show that manifestly sequential domains and functions form a cpo-enriched cartesian-closed category $\mathcal{SEQ}(\mathbb{E})$ provided $\mathbb{E} \neq \emptyset$. We begin by defining the category $\mathcal{SEQ}(\mathbb{E})$ consisting of (manifestly) sequential domains and functions relative to an error set $\mathbb{E}$.

Definition 6.1 ($\mathcal{SEQ}(\mathbb{E})$) The category of (manifestly) sequential objects and functions over an error set $\mathbb{E}$ is defined as follows:

1. the collection of objects is $\{\mathbb{D}(\mathbf{M}) \,|\, \mathbf{M} \text{ is an } sds\}$, the set of (manifestly) sequential domains over *sds*'s relative to $\mathbb{E}$;

2. the collection of arrows between the objects $\mathbb{D}(\mathbf{M}_1)$ and $\mathbb{D}(\mathbf{M}_2)$ is $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$,[14] the set of (manifestly) sequential functions relative to $\mathbb{E}$;

3. the composition operation for arrows is the usual function composition;

[14] This definition of homset is independent of the choice of the *sds*'s $\mathbf{M}_1$ and $\mathbf{M}_2$ representing $\mathbb{D}_1$ and $\mathbb{D}_2$, where $\mathbb{D}_1 = \mathbb{D}(\mathbf{M}_1)$ and $\mathbb{D}_2 = \mathbb{D}(\mathbf{M}_2)$.

4. for each object the identity arrow is the identity function.

We use the notation $\mathcal{SEQ}(\mathbb{E})$ to refer to this category. ∎

By Proposition 5.6, the preceding definition formalizes a category. More importantly, most of these categories, specifically, all those generated over non-empty error sets, are cartesian-closed categories.

Theorem 6.2 *1. If $\mathbb{E} \neq \emptyset$, then $\mathcal{SEQ}(\mathbb{E})$ is a cartesian-closed category.*

2. $\mathcal{SEQ}(\emptyset)$ is not cartesian-closed.

Proof. For non-empty sets of error values $\mathbb{E}$, we prove in the following two subsections that the category is cartesian (Lemma 6.6) and cartesian-closed (Lemma 6.22). Berry and Curien [4: Theorem 3.4.6] showed that the category of concrete domains and sequential functions is not cartesian-closed. In the Appendix, we prove that $\mathcal{SEQ}(\emptyset)$ is equivalent to the Berry and Curien's original category of (filiform) concrete domains and sequential functions, which implies the second part of the theorem. ∎

An alternative way of defining a cpo-enriched cartesian-closed category for SPCF exploits the bijection between an exponent object $\mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ in $\mathcal{SEQ}(\mathbb{E})$ (see Definition 6.21) and the corresponding manifestly sequential function space $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ when $\mathbb{E} \neq \emptyset$. The category $\mathcal{S}eq(\mathbb{E})$ has the same objects as $\mathcal{SEQ}(\mathbb{E})$ but it uses the set $\mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ instead of $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ as the homset $\mathbb{D}(M_1) \longrightarrow \mathbb{D}(M_2)$ where the composition of such algorithms is defined in terms of Berry and Curien's abstract algorithms (see Definition A.3). This category is important because it relates our work to the original work of Berry and Curien on sequential algorithms. When $\mathbb{E} = \emptyset$, $\mathcal{S}eq(\mathbb{E})$ contains more arrows than $\mathcal{SEQ}(\mathbb{E})$ because many functions are represented by more than one decision tree. The additional arrows make $\mathcal{S}eq(\mathbb{E})$ cartesian-closed at the cost of sacrificing extensionality. Appendix A defines the category $\mathcal{S}eq(\mathbb{E})$ and discusses its relationship to both $\mathcal{SEQ}(\mathbb{E})$ and Berry and Curien's original category.

Theorem 6.3 *$\mathcal{S}eq(\mathbb{E})$ is cartesian-closed for all sets of error values $\mathbb{E}$.*

Proof. Appendix A contains the equivalence proof between $\mathcal{S}eq(\emptyset)$ and the Berry-Curien cartesian-closed category of filiform concrete data structures. The rest follows from Theorem 6.2. ∎

The rest of this section is dedicated to proving the two lemmas cited in the proof of Theorem 6.2. The first subsection shows that the category $\mathcal{SEQ}(\mathbb{E})$ is cartesian. The second subsection defines the exponent of two *sds*'s, proves an extensionality theorem for elements of the exponent, and uses this result to prove that the category $\mathcal{SEQ}(\mathbb{E})$ is cartesian-closed if $\mathbb{E}$ is non-empty. In these two subsections, $\mathbb{E}$ is a fixed but unspecified, non-empty set of error values. The results of the first subsection hold for all possible error sets, but those of the second subsection are sensitive to the cardinality of $\mathbb{E}$.

6.1 $\mathcal{SEQ}(\mathbb{E})$ is Cartesian

We begin by identifying the terminal object in $\mathcal{SEQ}(\mathbb{E})$ and defining the product construction on objects of $\mathcal{SEQ}(\mathbb{E})$.

Definition 6.4 (*Terminal sds, Object, Arrow***)** The *terminal sds* is the triple $(\emptyset, \emptyset, \emptyset)$. The terminal object 1 is the domain containing the single element $\perp$ $(\emptyset)$. If A is an object in $\mathcal{SEQ}(\mathbb{E})$, then the arrow $1^A : A \longrightarrow 1$ is the constant function whose graph is $\{(x, \perp) \mid x \in A\}$. ∎

The construction of a product in $\mathcal{SEQ}(\mathbb{E})$ relies on the notion of a product *sds*, which is basically a "disjoint union" of two *sds*'s.

Definition 6.5 (*Product sds, Object, Arrow***)** Let $\mathbf{M}_1 = (A_1, D_1, P_1)$ and $\mathbf{M}_2 = (A_2, D_2, P_2)$ be two *sds*'s. The *product sds* of $\mathbf{M}_1$ and $\mathbf{M}_2$ is $\mathbf{M}_1 \times \mathbf{M}_2 = (A, D, P)$ where

$$
\begin{array}{rcl}
A & = & A_1 \uplus A_2 \\
D & = & D_1 \uplus D_2 \\
P & = & \{\langle b_1, i\rangle \cdot \ldots \cdot \langle b_n, i\rangle \mid b_1 \cdot \ldots \cdot b_n \in P_i \text{ for } i \in \{1, 2\}\}
\end{array}
$$

The *product object* of $\mathbb{D}(\mathbf{M}_1)$ and $\mathbb{D}(\mathbf{M}_2)$ is the domain $\mathbb{D}(\mathbf{M}_1 \times \mathbf{M}_2)$. The projection functions $\pi_i^\times : \mathbb{D}(\mathbf{M}_1 \times \mathbf{M}_2) \longrightarrow \mathbb{D}(\mathbf{M}_1)$ remove the disjoint union tag from each component of a path from the respective domain and ignore other paths:

$$
\begin{array}{rcl}
\pi_i^\times(x) & = & \{a_1 \cdot d_1 \cdot \ldots \cdot a_n \cdot d_n \mid \langle a_1, i\rangle \cdot \langle d_1, i\rangle \cdot \ldots \cdot \langle a_n, i\rangle \cdot \langle d_n, i\rangle \in x\} \\
& & \cup \\
& & \{a_1 \cdot d_1 \cdot \ldots \cdot a_n \cdot \mathsf{e} \mid \langle a_1, i\rangle \cdot \langle d_1, i\rangle \cdot \ldots \cdot \langle a_n, i\rangle \cdot \mathsf{e} \in x\}.
\end{array}
$$

If $\mathbb{D}(\mathbf{M})$ is an object and $f_i : \mathbb{D}(\mathbf{M}) \longrightarrow \mathbb{D}(\mathbf{M}_i)$, $i \in \{1, 2\}$, are functions, then the pair $\langle f_1, f_2\rangle$ is the function that combines the results of both functions:

$$\langle f_1, f_2\rangle(x) = inj_1(f_1(x)) \cup inj_2(f_2(x))$$

where the injection functions $inj_i : \mathbb{D}(\mathbf{M}_i) \longrightarrow \mathbb{D}(\mathbf{M}_1 \times \mathbf{M}_2)$, $i \in \{1, 2\}$, put an appropriate tag on each element of each path:

$$
\begin{array}{rcl}
inj_i(x) & = & \{\langle a_1, i\rangle \cdot \langle d_1, i\rangle \cdot \ldots \cdot \langle a_n, i\rangle \cdot \langle d_n, i\rangle \mid a_1 \cdot d_1 \cdot \ldots \cdot a_n \cdot d_n \in x\} \\
& & \cup \\
& & \{\langle a_1, i\rangle \cdot \langle d_1, i\rangle \cdot \ldots \cdot \langle a_n, i\rangle \cdot \mathsf{e} \mid a_1 \cdot d_1 \cdot \ldots \cdot a_n \cdot \mathsf{e} \in x\}.
\end{array}
$$

The second components in the unions defining the projection and the injection functions above are included to make the functions manifestly sequential. ∎

It is easy to verify that the preceding definition constitutes a product construction for $\mathcal{SEQ}(\mathbb{E})$. Specifically, the projection and pairing functions are manifestly sequential functions and satisfy the required equations.

Lemma 6.6 *Let $f_1 : \mathbb{D}(\mathbf{M}) \longrightarrow \mathbb{D}(\mathbf{M}_1)$ and $f_2 : \mathbb{D}(\mathbf{M}) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ be manifestly sequential functions. Then $\pi_1^\times$, $\pi_2^\times$, and $\langle f_1, f_2\rangle$ are manifestly sequential functions that satisfy the following equations:*

$$\begin{array}{rcl} \pi_1^\times \circ \langle f_1, f_2\rangle & = & f_1 \\ \pi_2^\times \circ \langle f_1, f_2\rangle & = & f_2 \\ \langle \pi_1^\times \circ f, \pi_2^\times \circ f\rangle & = & f \end{array}$$

where f ranges over $\mathbb{D}(\mathbf{M}) \longrightarrow \mathbb{D}(\mathbf{M}_1 \times \mathbf{M}_2)$.

Proof. The functions $\pi_1^\times$, $\pi_2^\times$, and $\langle f_1, f_2\rangle$ for appropriate f_1, f_2 are clearly monotonic and continuous. The sequentiality index of $\pi_i^\times$ for $(x, a_1 \cdot d_1 \cdot \ldots \cdot a_n)$ is $\langle a_1, i\rangle \cdot \langle d_1, i\rangle \cdot \ldots \cdot \langle a_n, i\rangle$. Similarly, the sequentiality index of the function $\langle f_1, f_2\rangle$ for $(x, \langle a'_1, i\rangle \cdot \langle d'_1, i\rangle \cdot \ldots \cdot \langle a'_n, i\rangle)$ is the sequentiality index of f_i for $(x, \cdot a'_1 d'_1 \ldots a'_n)$. The function is manifestly sequential because f_1 and f_2 are manifestly sequential.

The rest of the proof is a straightforward calculation based on the preceding definitions. For any x,

$$\begin{array}{rcl} (\pi_1^\times \circ \langle f_1, f_2\rangle)(x) & = & \pi_1^\times(inj_1(f_1(x)) \cup inj_2(f_2(x))) \\ & = & \pi_1^\times(inj_1(f_1(x))) \cup \pi_1^\times(inj_2(f_2(x))) \\ & = & \pi_1^\times(inj_1(f_1(x))) \\ & = & f_1(x). \end{array}$$

Hence, $\pi_1^\times \circ \langle f_1, f_2\rangle = f_1$. Similarly, for all x,

$$\begin{array}{rcl} (\langle \pi_1^\times \circ f, \pi_2^\times \circ f\rangle)(x) & = & inj_1(\pi_1^\times(f(x))) \cup inj_2(\pi_2^\times(f(x))) \\ & = & (f(x) \setminus inj_2(f(x))) \cup (f(x) \setminus inj_1(f(x))) \\ & = & f(x)\,, \end{array}$$

implying that $\langle \pi_1^\times \circ f, \pi_2^\times \circ f\rangle = f$. ∎

We have thus shown that $\mathcal{SEQ}(\mathbb{E})$ is a cartesian category.

6.2 $\mathcal{SEQ}(\mathbb{E})$ is Cartesian-Closed

The construction of exponent objects in $\mathcal{SEQ}(\mathbb{E})$ relies on the extensional representation of manifestly sequential functions as trees. The first part of this section explains that representation and identifies its important technical properties. The second part uses these properties to prove that $\mathcal{SEQ}(\mathbb{E})$ is cartesian-closed provided $\mathbb{E}$ is non-empty. For a general treatment of the case $\mathbb{E} = \emptyset$ we refer to Curien's monograph [8] and Appendix A.

6.2.1 An Extensional Representation of Functions as Trees

The informal description of the decision tree representation for functions in the preceding two sections referred to the *strategy* that a function uses to explore its argument. Roughly speaking, a strategy is a sequence of queries and responses between the function and its argument (which may be a tuple). The application of a function follows a particular strategy as long the argument yields the anticipated responses. If the actual response differs from the expected response, the function must use an alternate strategy (consistent with the explored prefix of the previous strategy) or diverge.

Technically, a strategy for a (finite) argument value x is a non-repetitive, alternating *sequence* of queries and responses, starting with a query, such that x is the collection of all responses in the sequence. In such a sequence, each response $q \cdot d$ is immediately preceded by the query q. Conversely, each query $p \cdot a$ is preceded (but not necessarily immediately) by the response p. These two properties capture the idea that a function can only explore a path in its argument after it has explored all approximations to the path. In addition, an *observable* response whose last element is an error value cannot appear in a strategy. Otherwise, a function could check for the presence of error values in its input and output a non-error value, violating the manifest sequentiality condition. If the exploration of the argument x encounters an error value, the application process must place the same error value in the output tree at the appropriate position. This part of the application process is encoded into the application function (see Definition 6.10 below). The formal definition of a strategy follows.[15]

Definition 6.7 (*Path Sequences***)** Let $\mathbf{M} = (A, D, P)$ be an *sds*. A path sequence s over $\mathbf{M}$ is a path $p_1 \cdot \ldots \cdot p_n$ over the alphabet $(Que_{\mathbf{M}} \cup Res_{\mathbf{M}})$ such that:

1. $s \in (Que_{\mathbf{M}}, Res_{\mathbf{M}})^*$, and s is non-repetitive in $Que_{\mathbf{M}}$ (which implies that the path is also non-repetitive in $Res_{\mathbf{M}}$);
2. for all $i \geq 1$ such that $2i+1 \leq |s|$, there exists $d \in D$ such that $s \,@\, (2i+1) = s \,@\, (2i) \cdot d$;
3. $s \,@\, 0 \in A$ (that is, $s \,@\, 0$ has length 1); and
4. for all $i \geq 1$ such that $2i \leq |s|$ there exists an j such that $2j+1 < 2i$ and $s \,@\, (2i) = s \,@\, (2j+1) \cdot a$ for some $a \in A$.

∎

A path sequence over $\mathbf{M}$ is a path constructed from tokens that are paths. For this reason, a path sequence can be interpreted as the linearization of a tree. This idea is formalized in the next lemma.

Lemma 6.8 *If s is a path sequence over $\mathbf{M}$, then* $\{s \,@\, (2i+1) \mid 2i+1 \leq |s|\} \in \mathbb{D}(M)$.

Proof. Let T stand for the set in question. We must show that T is prefix-closed and glb-closed. By conditions 2 and 4 in the preceding definition, and by induction on the position

[15] The set of path sequences over $\mathbf{M}$ can be formulated as an *sds* with $Que_{\mathbf{M}}$ and $Res_{\mathbf{M}}$ as the sets of addresses and responses. This observation is the basis for a factorization of the function space construction into a construction of path sequences and an *affine* function space. One can show that the two ways of constructing the function space yield isomorphic results, an idea that naturally leads to the construction of a model of affine and possibly linear logic. This decomposition is explored in forthcoming papers by Lamarche [20] and by the second author [11].

in s, T contains all response predecessors of its elements, *i.e.*, T is prefix-closed. Next assume that $q \cdot d, q \cdot d' \in T$. Then $q \cdot d = s \,@\, (2i+1)$ for some i and $q \cdot d' = s \,@\, (2j+1)$ for some j. By condition 2, $q = s \,@\, (2i) = s \,@\, (2j)$, and $i = j$ by condition 1. This implies that $d = d'$. Hence, T is also glb-closed, proving the lemma. ∎

Notation Given the preceding lemma, it makes sense to introduce the notation $\|s\|$ (pronounced "tree of s") to denote the tree $\{s \,@\, (2i+1) \mid 2i+1 \leq |s|\}$.

After a successful (possibly void) exploration of its argument based on some strategy, the function may construct a path in its output. Thus, a path in an element of an exponent object alternates between pieces of a strategy and pieces of output information. Since an output path, like any other path, is an alternating path of addresses and data, a path in a "functional tree" is a sequence starting with an address a on an output path, continued with a strategy, eventually followed by a datum that could be associated with a on the output path. The path can then continue with a new output address, followed by a (portion of a) strategy eventually followed by an output datum, and so on. This representation of functions is formalized in the following construction.

Definition 6.9 (*Exponent sds***)** Let $\mathbf{M}_1 = (A_1, D_1, P_1)$ and $\mathbf{M}_2 = (A_2, D_2, P_2)$ be *sds*'s. Let $Res_1 = Res_{\mathbf{M}_1}$, $Que_1 = Que_{\mathbf{M}_1}$, and S_1 be the set of path sequences over $\mathbf{M}_1$. The *exponent sds* $\mathbf{M}_1 \Rightarrow \mathbf{M}_2$ is (A, D, P) where

$$
\begin{array}{rcll}
A & = & Res_1 \uplus A_2 & \\
D & = & Que_1 \uplus D_2 & \\
P & = & \{p \in (A,D)^* \mid & \pi_1^{\Rightarrow}(p) \in S_1, \pi_2^{\Rightarrow}(p) \in P_2 \text{ or } \pi_2^{\Rightarrow}(p) = \epsilon \\
 & & & p \,@\, 0 = \langle a, 2 \rangle, a \in A_2, \\
 & & & \text{if } p \,@\, (i+1) = \langle a, 2 \rangle, a \in A_2 \text{ then } p \,@\, i = \langle d, 2 \rangle, d \in D_2, \\
 & & & \text{if } p \,@\, (i+1) = \langle r, 1 \rangle, r \in Res_1 \text{ then } p \,@\, i = \langle q, 1 \rangle, q \in Que_1\}
\end{array}
.
$$

The corresponding recursive functions $\pi_1^{\Rightarrow} : P \longrightarrow (Que_1 \cup Res_1)^*$ and $\pi_2^{\Rightarrow} : P \longrightarrow (A_2 \cup D_2)^*$ are defined as follows:

$$
\pi_i^{\Rightarrow}(\epsilon) = \epsilon; \qquad \pi_i^{\Rightarrow}(p \cdot \langle x, i \rangle) = \pi_i^{\Rightarrow}(p) \cdot x; \qquad \pi_i^{\Rightarrow}(p \cdot \langle x, j \rangle) = \pi_i^{\Rightarrow}(p) \text{ if } i \neq j
$$

for $x \in A_2 \cup D_2 \cup Res_1 \cup Que_1$ and $i, j \in \{1, 2\}$. The extensions of these functions to paths in domain elements propagate errors:

$$
\pi_i^{\Rightarrow}(p \cdot \mathsf{e}) = \pi_i^{\Rightarrow}(p) \cdot \mathsf{e}.
$$

Note: In the set comprehension defining P, the two last conditions imply each other; they are both mentioned for clarity. ∎

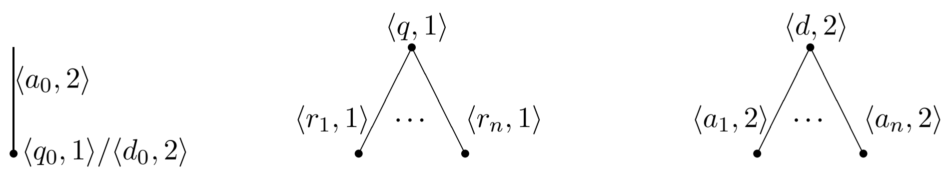
To ensure that the exponent construction is well-defined, we need to confirm that P is a valid set of paths. First, it does not contain the empty path since $\pi_2^{\Rightarrow}(p) \in P_2$ forces $p \neq \epsilon$. Second, it is prefix-closed because S_1 and P_2 are prefix-closed. Note that the construction precisely captures the idea that a procedure cannot examine a node in a decision tree unless it has already examined its ancestors. Technically, if $p \cdot \langle q, 1 \rangle$ is a response in the exponent, then $q \in Open(\|\pi_1^{\Rightarrow}(p)\|)$.

A simple example of an exponent *sds* is $\mathbf{N} \Rightarrow \mathbf{N}$:

$$\begin{array}{rcl} A & = & \{\langle ?, 2\rangle, \langle ? \cdot n, 1\rangle \,|\, n \in \mathbb{N}\} \\ D & = & \{\langle n, 2\rangle, \langle ?, 1\rangle \,|\, n \in \mathbb{N}\} \\ P & = & \left\{ \begin{array}{l} \langle ?, 2\rangle,\ \langle ?, 2\rangle \cdot \langle n, 2\rangle,\ \langle ?, 2\rangle \cdot \langle ?, 1\rangle \\ \langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot n, 1\rangle,\ \langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot n, 1\rangle \cdot \langle m, 2\rangle \end{array} \middle|\, m, n \in \mathbb{N} \right\} . \end{array}$$

Similarly, the *sds* $\mathbf{N}^4 \to \mathbf{N}$ described in the previous section is an exponent modulo the renaming of addresses and data.

The shape of a tree over an exponent *sds* satisfies three simple constraints, which we exploit in drawing trees schematically:



Top of a tree — Query and Responses — Datum and Addresses

The top of the tree is always an initial address from $\mathbf{M}_2$ followed by a either a datum from $\mathbf{M}_2$ or a one-element query from $\mathbf{M}_1$. The root of a proper subtree may be labeled with a query *q*. It may then have as many sons as there are legal responses r_1 through r_n; the responses are used as labels for the outgoing edges from a query node. Similarly, a node may be labeled with a datum *d* from $\mathbf{M}_2$ and the outgoing edges may then be labeled with as many addresses a_1 through a_n from $\mathbf{M}_2$ as can possibly follow on a legal path in $\mathbf{M}_2$. The trees in Section 4 omit the top part, which can always be restored, and do not require the datum/address shape because the corresponding functions are always fully uncurried.

Figure 3 shows left-addition as an example of a tree in $\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$. A more interesting example of a function tree over $(\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$, including its SPCF text, is displayed in Figure 4. The tree tests whether its input is a strict function, and if so returns the function `sub1` when the argument maps 0 to 1 and 1 to 2. For non-strict inputs, the function returns whatever the input returns.

A path *p* in a decision tree of an exponent *sds* has a natural interpretation as a piece of an algorithm. Such a path has the shape:

$$\langle a_1, 2\rangle \cdot s_1^* \cdot \langle d_1, 2\rangle \cdot \langle a_2, 2\rangle \cdot \ldots \cdot \langle d_{n-1}, 2\rangle \cdot \langle a_n, 2\rangle \cdot s_n^* \cdot \langle d_n, 2\rangle \cdot \langle a_{n+1}, 2\rangle \cdot s_{n+1}^*$$

where each segment s_i^*, for $1 \le i \le n$, is a tagged portion of a strategy:

$$s_i^* = \langle q_{i,1}, 1\rangle \cdot \langle r_{i,1}, 1\rangle \cdot \ldots \cdot \langle q_{i,m_i}, 1\rangle \cdot \langle r_{i,m_i}, 1\rangle$$

for some $m_i \ge 0$. When stripped of its tags, each s_i^* is a portion

$$s_i = q_{i,1} \cdot r_{i,1} \cdot \ldots \cdot q_{i,m_i} \cdot r_{i,m_i}$$

of a path sequence such that for all $i, 1 \le i \le n+1$, the composite sequence $s_1 \ldots s_i$ is a path sequence (over $\mathbf{M}_1$). For example the right-most path in the tree of Figure 4, the

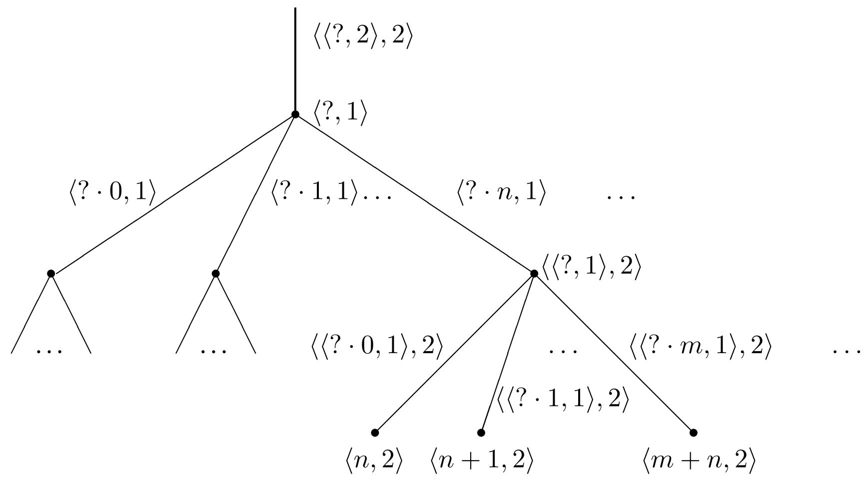


Figure 3: Left Addition: $+_l$

pieces are:

$$
\begin{array}{rcl}
\langle a_1, 2\rangle & = & \langle\langle ?, 2\rangle, 2\rangle \\
s_1^* & = & \begin{array}{ll}
\text{query} & \text{response} \\
\hline
\langle\langle ?, 2\rangle, 1\rangle & \langle\langle ?, 2\rangle \cdot \langle ?, 1\rangle, 1\rangle \\
\langle\langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot 0, 1\rangle, 1\rangle & \langle\langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot 0, 1\rangle \cdot \langle 1, 2\rangle, 1\rangle \\
\langle\langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot 1, 1\rangle, 1\rangle & \langle\langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot 1, 1\rangle \cdot \langle 2, 2\rangle, 1\rangle
\end{array} \\
\langle d_1, 2\rangle & = & \langle\langle ?, 1\rangle, 2\rangle \\
\langle a_2, 2\rangle & = & \langle\langle ? \cdot n, 1\rangle, 2\rangle \\
s_2^* & = & \epsilon \\
\langle d_2, 2\rangle & = & \langle\langle (n-1), 2\rangle, 2\rangle \\
s_3^* & = & \epsilon
\end{array}
$$

With the initial address a_1, the path *p* announces that the function attempts to construct an output path starting with a_1. If it can now successfully explore the argument according to strategy s_1, the function will add the datum d_1 to the output path, finishing the first possible approximation to the response. Then the process starts over with the announcement that the function will place a datum at address a_2 if the strategy $s_1 s_2$ can be pursued successfully, and so on. If at any point in the exploration of the argument a match yields no value at all, the process stops. Similarly, if the match produces an error value, the function must place the same error value on the output path behind the "announced" address and then the process terminates. The following definition formalizes the idea of "applying" a tree from the domain over an exponent *sds* to an argument tree.

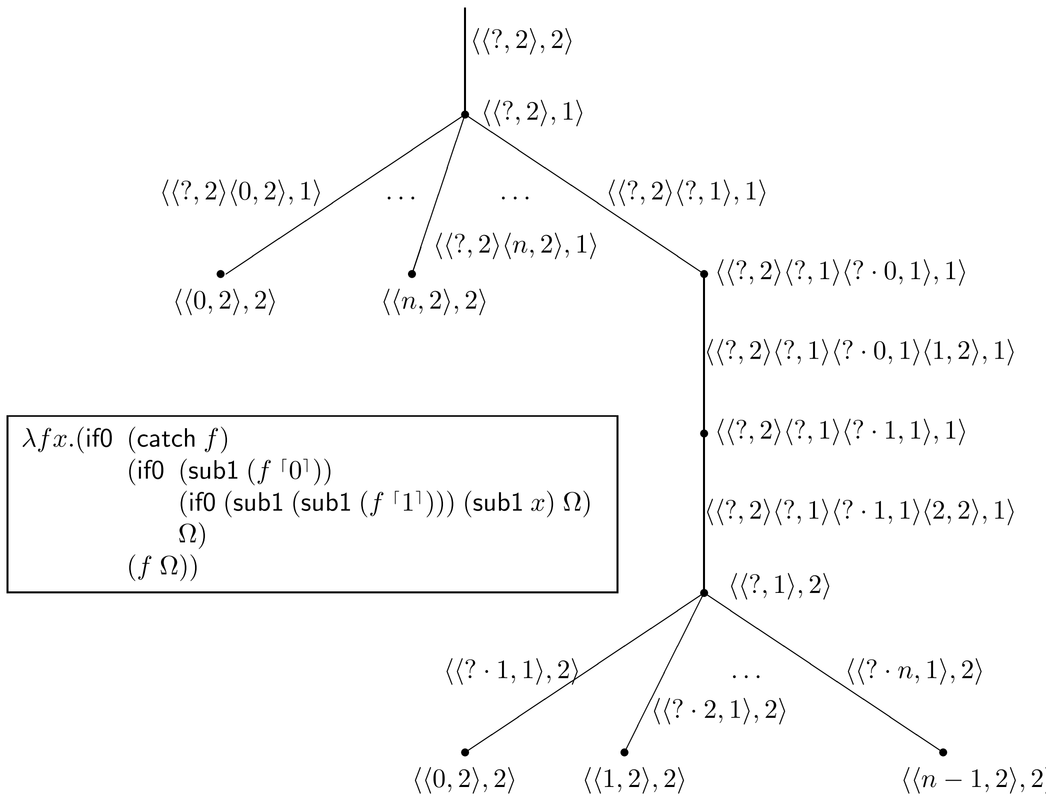


Figure 4: A Tree in $((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}))$

Definition 6.10 (*Application***)** Let $\mathbf{M}_1 \Rightarrow \mathbf{M}_2$ be the exponent *sds* of $\mathbf{M}_1$ and $\mathbf{M}_2$. The *application* of some tree $t \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ to some tree $x \in \mathbb{D}(\mathbf{M}_1)$, written as $t \star x$, is defined as follows:

$$
\begin{array}{rl}
t \star x = & \{\pi_2^{\Rightarrow}(p) \in Res_{\mathbf{M_2}} \mid \|\pi_1^{\Rightarrow}(p)\| \sqsubseteq x, p \in t\} \\
\cup & \{\pi_2^{\Rightarrow}(p) \cdot \mathsf{e} \in Res^e_{\mathbf{M_2}} \mid \|\pi_1^{\Rightarrow}(p)\| \cup \{q \cdot \mathsf{e}\} \sqsubseteq x,\ p \cdot \langle q, 1\rangle \in t\} \\
\cup & \{\pi_2^{\Rightarrow}(p) \cdot \mathsf{e} \in Res^e_{\mathbf{M_2}} \mid \|\pi_1^{\Rightarrow}(p)\| \sqsubseteq x,\ p \cdot \mathsf{e} \in t\}.
\end{array}
$$

∎

The formal definition of application captures the intuitive ideas presented in the preceding sections about manifestly sequential functions. When a decision tree is interpreted as a function, it inspects the argument tree one step at a time, with missing information resulting in a bottom output and erroneous information being propagated. It constructs distinct paths in the output tree independently, but each path is constructed sequentially. If a path occurs in the output, it is due to precisely one path.

Lemma 6.11 *If $t \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ and $x \in \mathbb{D}(\mathbf{M}_1)$, then $t \star x \in \mathbb{D}(\mathbf{M}_2)$.*

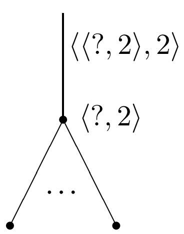
Proof. Each element of $t \star x$ is clearly an observable response over $\mathbf{M}_2$ by the definition of the application operator. Based on the fact that t is prefix- and glb-closed, it is easy to verify that $t \star x$ satisfies the same properties. ∎

We can now show that application is "functional", *i.e.*, for fixed t and varying x, $t \star x$ uniquely determines t. This extensionality property holds only when the set of errors $\mathbb{E}$ is non-empty.

Theorem 6.12 (Extensionality) *Let $t, t' \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$. Then:*

1. *if $\mathbb{E} = \emptyset$, $t \star x = t' \star x$ for all $x \in \mathbb{D}(\mathbf{M}_1)$ does* not *imply that $t = t'$;*
2. *if $\mathbb{E}$ contains a single element, $t = t'$ iff $t \star x = t' \star x$ for all $x \in \mathbb{D}(\mathbf{M}_1)$;*
3. *if $\mathbb{E}$ contains more than one element, $t \sqsubseteq t'$ iff $t \star x \sqsubseteq t' \star x$ for all $x \in \mathbb{D}(\mathbf{M}_1)$.*

Proof. Part 1 is obvious: the trees for the two addition procedures $+_l$ and $+_r$ introduced in Section 3 are distinct but they compute the same sequential function over the natural numbers. The tree for $+_l$ is given in Figure 3 in Subsection 6.2.1. In contrast, the tree for $+_r$ has the following form:



For the remaining cases, the left to right direction is obvious. For the converse, we first prove Part 3 by contra-position and return to Part 2 below. Assume $t \not\sqsubseteq t'$, *i.e.*, there exists p such that $p \in t$ and $p \notin t'$. Let p^* be the maximal path approximating p that also approximates some $p' \in t'$. To prove the claim we will derive a witness x from p^* such that $t \star x \not\sqsubseteq t' \star x$. Clearly, the witness must dominate $x_0 = \|\pi_1^{\Rightarrow}(p^*)\|$. To determine the rest of the witness, we need to distinguish two cases:

1. p^* is empty or is a response: Since p^* is maximal in t' in the direction of p, it suffices to understand how p extends p^* in order to construct the witness:

path in t	witness x	path in $t \star x$, not in $t' \star x$
$p^* \cdot \langle r, 1\rangle \cdot \langle q, 1\rangle$	$x = x_0 \cup \{r, q \cdot \mathsf{e}\}$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$
$p^* \cdot \langle r, 1\rangle \cdot \langle d, 2\rangle$	$x = x_0 \cup \{r\}$	$\pi_2^{\Rightarrow}(p^*) \cdot d$
$p^* \cdot \langle r, 1\rangle \cdot \mathsf{e}$	$x = x_0 \cup \{r\}$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$
$p^* \cdot \langle a, 2\rangle \cdot \langle q, 1\rangle$	$x = x_0 \cup \{q \cdot \mathsf{e}\}$	$\pi_2^{\Rightarrow}(p^*) \cdot a \cdot \mathsf{e}$
$p^* \cdot \langle a, 2\rangle \cdot \langle d, 2\rangle$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot a \cdot d$
$p^* \cdot \langle a, 2\rangle \cdot \mathsf{e}$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot a \cdot \mathsf{e}$

Given a witness, it is easy to prove that the path in the last column is indeed in $t \star x$ but not in $t' \star x$. For an example, consider the first line. If $\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e} \in t' \star x$ then either

- there exists a response $r' \in t'$ such that $\pi_2^{\Rightarrow}(r') = \pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$ and $\|\pi_1^{\Rightarrow}(r')\| \sqsubseteq x$ (error generation); or
- there exists a response $r' = p' \cdot \langle q', 1\rangle \in t'$ such that $\|\pi_1^{\Rightarrow}(p')\| \cup \{q' \cdot \mathsf{e}\} \sqsubseteq x$ and $\pi_2^{\Rightarrow}(p^*) = \pi_2^{\Rightarrow}(p')$ (error propagation).

In either case, $\|\pi_1^{\Rightarrow}(p^*)\| \uparrow \|\pi_1^{\Rightarrow}(r')\|$ and the outputs are comparable, so that p^* and r' are comparable as well by the following lemma (6.13). First, assume $p^* \sqsubseteq r'$. Since $\|\pi_1^{\Rightarrow}(r')\| \sqsubseteq x$, it is moreover true that $p^* \cdot \langle r, 1\rangle \sqsubseteq r'$ because x can only respond in one way to the question at the end of p^*. But this contradicts the maximality of p^* in t' in the direction of p. Second, assume $r' \sqsubseteq p^*$, which is only possible in the error propagation case since $\pi_2^{\Rightarrow}(p^*) \sqsubset \pi_2^{\Rightarrow}(r')$ in the error generation case. Now, for some $r \in Res_{\mathbf{M}_2}$, $r' \cdot \langle r, 1\rangle \sqsubseteq p^*$, which implies that $r \in \|\pi_1^{\Rightarrow}(p^*)\| \sqsubseteq x$ and thus contradicts $q' : \mathsf{e} \in x$. — The proofs for other cases proceed in similar fashion.

2. p^* is a query: Now p^* has extensions in both t and t' and the subcases accordingly account for this fact:

path in t	path in t'	witness x	path in $t \star x$, not in $t' \star x$	
$p^* \cdot \langle q, 1\rangle$	$p^* \cdot \langle q', 1\rangle$	$x = x_0 \cup \{q \cdot \mathsf{e}\}$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$	
where $q \neq q'$				
$p^* \cdot \langle q, 1\rangle$	$p^* \cdot \langle d', 2\rangle$	$x = x_0 \cup \{q \cdot \mathsf{e}\}$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$	
$p^* \cdot \langle q, 1\rangle$	$p^* \cdot \mathsf{e}'$	$x = x_0 \cup \{q \cdot \mathsf{e} \,\vert\, \mathsf{e} \neq \mathsf{e}'\}$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$	(†)
$p^* \cdot \langle d, 2\rangle$	$p^* \cdot \langle q', 1\rangle$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot d$	
$p^* \cdot \langle d, 2\rangle$	$p^* \cdot \langle d', 2\rangle$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot d$	
where $d \neq d'$				
$p^* \cdot \langle d, 2\rangle$	$p^* \cdot \mathsf{e}'$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot d$	
$p^* \cdot \mathsf{e}$	$p^* \cdot \langle q', 1\rangle$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$	
$p^* \cdot \mathsf{e}$	$p^* \cdot \langle d', 2\rangle$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$	
$p^* \cdot \mathsf{e}$	$p^* \cdot \mathsf{e}'$	$x = x_0$	$\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}$	
where $\mathsf{e} \neq \mathsf{e}'$				

Again, we only illustrate with an example the kind of proofs that are required to show that the path in the last column is not in $t' \star x$. Consider the fourth line. For $t' \star x$ to contain $\pi_2^{\Rightarrow}(p^*) \cdot d$, there must be a response $r' \in t'$ such that $\pi_2^{\Rightarrow}(r') = \pi_2^{\Rightarrow}(p^*) \cdot d$ and $\|\pi_1^{\Rightarrow}(r')\| \sqsubseteq x$. By Lemma 6.13, $p^* \cdot \langle d, 2\rangle \sqsubseteq r'$ since $\pi_2^{\Rightarrow}(p^*) \sqsubset \pi_2^{\Rightarrow}(r')$. Then we should have that $p^* = p^* \cdot \langle q', 1\rangle \sqcap p^* \cdot \langle d, 2\rangle$ is in t'. But p^* is a query and thus cannot be a member of the tree t'. Hence, $\pi_2^{\Rightarrow}(p^*) \cdot d \notin t' \star x$.

The proof of Part 2 is similar to the proof of Part 3, except for the case marked with (†) because there is no second element in $\mathbb{E} = \{\mathsf{e}'\}$ that can be used to extend q. Thus, for example, in $\mathbf{N} \Rightarrow \mathbf{N}$, $\langle ?, 2\rangle \cdot \langle ?, 1\rangle \not\sqsubseteq \langle ?, 2\rangle \cdot \mathsf{e}'$ yet $\langle ?, 2\rangle \cdot \langle ?, 1\rangle \star x = \emptyset \sqsubseteq \langle ?, 2\rangle \cdot \mathsf{e}' \star x$ for any x. But the following equivalent formulation of part 2 holds:

2′. If $\mathbb{E}$ contains a single element, $t \not\sqsubseteq t'$ implies $t \star x \neq t' \star x$, for some $x \in \mathbb{D}(\mathbf{M}_1)$.

With the exception of the case marked (†), all cases work as before. For case (†), $\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}' \in t' \star x_0$ but $\pi_2^{\Rightarrow}(p^*) \cdot \mathsf{e}' \notin t \star x_0$, *i.e.*, $t' \star x_0 \not\sqsubseteq t \star x_0$. ∎

To complete the proof of the extensionality theorem, we need to verify the following lemma, which we use again later.

Lemma 6.13 *Let $t \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ and let $p, p' \in t$. If $\|\pi_1^{\Rightarrow}(p)\| \uparrow \|\pi_1^{\Rightarrow}(p')\|$ and $\pi_2^{\Rightarrow}(p) \sqsubseteq \pi_2^{\Rightarrow}(p')$, then $p \sqsubseteq p'$ or $p' \sqsubseteq p$. Moreover, if $\pi_2^{\Rightarrow}(p) \sqsubset \pi_2^{\Rightarrow}(p')$ then $p \sqsubset p'$.*

Proof. Assume $\|\pi_1^{\Rightarrow}(p)\|$ and $\|\pi_1^{\Rightarrow}(p')\|$ are consistent and $\pi_2^{\Rightarrow}(p) \sqsubseteq \pi_2^{\Rightarrow}(p')$. Let $s = p \sqcap p'$. Assume that neither $s = p$ nor $s = p'$, *i.e.*, p, p' are incomparable, and consider the following cases:

$s = s' \cdot \langle q, 1\rangle$· Then for some r and r', $s \cdot \langle r, 1\rangle \sqsubseteq p$ and $s \cdot \langle r', 1\rangle \sqsubseteq p'$ with $r \neq r'$. But this contradicts the consistency of $\|\pi_1^{\Rightarrow}(p)\|$ and $\|\pi_1^{\Rightarrow}(p')\|$.

$s = \epsilon$ **or** $s = s' \cdot \langle d, 2\rangle$· Here, for some addresses a, a', $s \cdot \langle a, 2\rangle \sqsubseteq p$ and $s \cdot \langle a', 2\rangle \sqsubseteq p'$ with $a \neq a'$. But again, this leads to a contradiction: if this were true, $\pi_2^{\Rightarrow}(p)$ could not possibly approximate $\pi_2^{\Rightarrow}(p')$.

Since both cases turn out to be impossible, p and p' must be comparable. If in addition $\pi_2^{\Rightarrow}(p) \sqsubset \pi_2^{\Rightarrow}(p')$ then clearly, $p \sqsubset p'$ by the definition of the projection function. ∎

The extensionality theorem confirms that each tree uniquely represents a function. It gives rise to the definition of a mapping from the domain over the exponent *sds* to the set of functions between the respective domains.

Definition 6.14 (Fun**)** Let $t \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$. Then $\mathsf{Fun}(t) = \underline{\lambda} x : \mathbb{D}(\mathbf{M}_1) \,.\, t \star x$. ∎

In fact, for a tree t, $\mathsf{Fun}(t)$ is always a manifestly sequential function.

Lemma 6.15 $\mathsf{Fun} : \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2) \longrightarrow \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$

Proof. Let $t \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$. Clearly, $\mathsf{Fun}(t)$ is a monotonic and continuous function. As for sequentiality, assume that for some x and for some $z \sqsupseteq x$, $q' \in \mathit{Open}(t \star x)$ and $q' \in \mathit{Answered}(t \star z)$. Let $q' = r' \cdot a$ (where, possibly, $r' = \epsilon$) and let $q' \cdot d_z \in t \star z$ for some d_z from $\mathbf{M}_2$. By the definition of the application operator, there must be

1. $p_z \in t$ such that $\pi_2^{\Rightarrow}(p_z) = r' \cdot a \cdot d_z$ and $\|\pi_1^{\Rightarrow}(p_z)\| \sqsubseteq z$; or,
2. $q_z \cdot \langle q_e, 1\rangle \in t$ such that $\pi_2^{\Rightarrow}(q_z) \cdot \mathsf{e} = r' \cdot a \cdot d_z$ and $\|\pi_1^{\Rightarrow}(q_z)\| \cup \{q_e \cdot \mathsf{e}\} \sqsubseteq z$.

The rest of the proof concentrates on the first case; the arguments can easily be modified for the second one.

Choose p and q_x such that

$$p \cdot \langle a, 2\rangle \cdot \ldots \cdot \langle q_x, 1\rangle \cdot \ldots \cdot \langle d_z, 2\rangle = p_z,$$

where $q_x \in \mathit{Open}(x)$ is the first such open query. The path fragment between $\langle a, 2\rangle$ and $\langle d_z, 2\rangle$ must contain a query $q_x \in \mathit{Open}(x)$. Otherwise, $\|\pi_1^{\Rightarrow}(p \cdot \langle a, 2\rangle \cdot \ldots \cdot \langle d_z, 2\rangle)\| \sqsubseteq x$ and $\pi_2^{\Rightarrow}(p \cdot \langle a, 2\rangle \cdot \ldots \cdot \langle d_z, 2\rangle) = q' \cdot d_z \in t \star x$, which contradicts $q' \in \mathit{Open}(t \star x)$.

It is easy to show that $\|\pi_1^{\Rightarrow}(p)\| \sqsubseteq x$. If $p \neq \epsilon$, then $\pi_2^{\Rightarrow}(p) = r'$, and $r' \neq \epsilon$. Since $r' \in t \star x$, there must be some path p' such that $\pi_2^{\Rightarrow}(p') = r'$ and $\|\pi_1^{\Rightarrow}(p')\| \sqsubseteq x$. By Lemma 6.13, p' and p are comparable. If $p' \sqsubset p$, then $\pi_2^{\Rightarrow}(p') \sqsubset r'$ because p ends in $\langle d, 2\rangle$ when r' ends in d; otherwise, if $p \sqsubset p'$, then $r' \cdot a \sqsubseteq \pi_2^{\Rightarrow}(p')$. Thus, $p' = p$. If $p = \epsilon$, then $\|\pi_1^{\Rightarrow}(p)\| = \perp \sqsubseteq x$.

The query q_x is the sequentiality index of $\mathsf{Fun}(t)$ for (x, q'). To prove this claim, let $y \in \mathbb{D}(\mathbf{M}_1)$ be such that $y \sqsupseteq x$ and $q' \in \mathit{Answered}(t \star y)$. By the same arguments as for *z*, *t* must contain some path p_y with

$$p \cdot \langle a, 2\rangle \cdot \ldots \cdot \langle q_y, 1\rangle \cdot \ldots \cdot \langle d_y, 2\rangle = p_y$$

for some first $q_y \in \mathit{Open}(x)$. The path *p* is the prefix for both p_z and p_y because at most one path in *t* can produce the output path r'. It follows from a simple inductive argument that any query following $\langle a, 2\rangle$ in front of q_y on p_y must be identical to the respective query on p_z due to the tree structure of *t* and that the following responses must be identical because they are in *x*.

Since all queries between $\langle a, 2\rangle$ and q_y are answered in *x*, all corresponding responses are identical to the respective responses on p_z. Since *t* is a tree, the queries must be identical, too. Hence, $q_y = q_x$, which proves that q_x is the sequentiality index.

The manifest sequentiality of $\mathsf{Fun}(t)$ follows immediately from the preceding portion of the proof and the definition of application. ∎

The proof of the preceding lemma also indicates how to find a tree from a manifestly sequential function f. Every path in the functional tree corresponding to f contributes one path to an output, if any, by exploring some finite piece of the input according to the strategy of f. The strategy is an alternating sequence of (unique) sequentiality indices and their responses mixed with output addresses and data. Collecting all of the paths for a function f in a set yields a tree.

Definition 6.16 (Tree, $\mathsf{Path}_f(x, p)$**)** Let $f : \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ be a sequential and manifestly sequential function between the domains $\mathbb{D}(\mathbf{M}_1)$ and $\mathbb{D}(\mathbf{M}_2)$. Let x be in $\mathbb{D}_0(\mathbf{M}_1)$, and let $p = a_1 d_1 \ldots a_n d_n$ be in $f(x)$ with, possibly, $d_n \in \mathbb{E}$.

The finite element $\mathsf{Path}_f(x, p) = \{p_0, \ldots, p_m\}$ is inductively defined as follows:

1. $p_0 = \begin{cases} \langle a_1, 2\rangle \cdot \langle d_1, 2\rangle & \text{if } a_1 \cdot d_1 \in f(\perp) \\ \langle a_1, 2\rangle \cdot \mathsf{e} & \text{if } a_1 \cdot \mathsf{e} = p \text{ and } p \in f(\perp) \\ \langle a_1, 2\rangle \cdot \langle \mathsf{si}_f(\perp, a_1), 1\rangle & \text{otherwise} \end{cases}$

2. if $p_i = q_i \cdot \langle q, 1\rangle$, $\pi_2^{\Rightarrow}(p_i) = a_1 \cdot d_1 \cdot \ldots \cdot a_j$ and there is an $r = q \cdot d$ from $\mathit{Res}_{\mathbf{M}_2}$ in x then

$$p_{i+1} = \begin{cases} q_{i+1} \cdot \langle d_j, 2\rangle & \text{if } \pi_2^{\Rightarrow}(p_i) \cdot d_j \in f(\|\pi_1^{\Rightarrow}(q_{i+1})\|) \\ q_{i+1} \cdot \mathsf{e} & \text{if } \pi_2^{\Rightarrow}(p_i) \cdot \mathsf{e} \in f(\|\pi_1^{\Rightarrow}(q_{i+1})\|) \\ q_{i+1} \cdot \langle \mathsf{si}_f(\|\pi_1^{\Rightarrow}(q_{i+1})\|, \pi_2^{\Rightarrow}(q_{i+1})), 1\rangle & \text{otherwise} \end{cases}$$

   where $q_{i+1} = p_i \cdot \langle r, 1\rangle$;

3. if $p_i = q_i \cdot \langle d_j, 2\rangle$, $\pi_2^{\Rightarrow}(p_i) = a_1 \cdot d_1 \cdot \ldots \cdot a_j \cdot d_j$, and $j < n$ then

$$p_{i+1} = \begin{cases} q_{i+1} \cdot \langle d_{j+1}, 2\rangle & \text{if } \pi_2^{\Rightarrow}(q_{i+1}) \cdot d_{j+1} \in f(\|\pi_1^{\Rightarrow}(p_i)\|) \\ q_{i+1} \cdot \mathsf{e} & \text{if } \pi_2^{\Rightarrow}(q_{i+1}) \cdot \mathsf{e} \in f(\|\pi_1^{\Rightarrow}(p_i)\|) \\ q_{i+1} \cdot \langle \mathsf{si}_f(\|\pi_1^{\Rightarrow}(p_i)\|, \pi_2^{\Rightarrow}(q_{i+1})), 1\rangle & \text{otherwise} \end{cases}$$

where $q_{i+1} = p_i \cdot \langle a_{j+1}, 2\rangle$.

The tree for f is the lub of all such finite elements:

$$\mathsf{Tree}(f) = \bigsqcup \{\mathsf{Path}_f(x,p) \mid x \in \mathbb{D}_0(\mathbf{M}_1), p \in f(x)\}.$$

∎

The proof that $\mathsf{Tree}(f)$ is a well-defined element of $\mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ requires a simple lemma about $\mathsf{Path}_f(x,p)$.

Lemma 6.17 *Let $f \in \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$, let $x \in \mathbb{D}_0(\mathbf{M}_1)$, and let $p \in f(x)$. Then*

1. $\mathsf{Path}_f(x,p) \in \mathbb{D}_0(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$*;*
2. $\|\pi_1^{\Rightarrow}(\mathsf{Path}_f(x,p))\|$ *is the minimal finite element* $x' \sqsubseteq x$ *such that* $p \in f(x')$*, and hence,* $p \in \mathsf{Path}_f(x,p) \star x$*.*

Proof. For Part 1, first note two invariants of the elements $p_0, \ldots, p_m$ in $\mathsf{Path}_f(x,p)$:

1. $p_i \in Res_{\mathbf{M}_1 \Rightarrow \mathbf{M}_2}$ for all *i*;
2. if $p_i, p_{i+1} \in \mathsf{Path}_f(x,p)$ then p_i is the immediate predecessor of p_{i+1}.

Hence, $\mathsf{Path}_f(x,p)$ is a prefix-closed chain of paths, and therefore is an element. By the finiteness of *p* and *x*, $\mathsf{Path}_f(x,p)$ is finite.

Part 2 first claims that there is a unique input threshold for every output path. The proof follows Berry's proof that sequential functions are stable [8:Proposition 2.4.7]. If $x \uparrow y$ then $f(x \sqcap y) = f(x) \sqcap f(y)$: Assume $x \uparrow y$. Clearly $f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$. Thus, assume that

$$f(x \sqcap y) \sqsubset f(x) \sqcap f(y).$$

This means that there exists a path p such that $p \in f(x)$, $p \in f(y)$, and $p \notin f(x \sqcap y)$. Let q' be the first open query for $f(x \sqcap y)$ below p. By the sequentiality of *f*,

$$q = \mathsf{si}_f(x \sqcap y, q')$$

is defined. By assumption, there are d_x and d_y such that $q \cdot d_x \in x$ and $q \cdot d_y \in y$. Since $x \uparrow y$, $d_x = d_y$. But therefore $q \cdot d_x \in x \sqcap y$, which contradicts $q \in Open(x \sqcap y)$. Hence, $f(x \sqcap y) = f(x) \sqcap f(y)$.

"Stability" implies the existence of a unique threshold: By continuity there exists a finite $x_0 \sqsubseteq x$ such that $p \in f(x_0)$. Let x_0, x_1 be two minimal elements such that $p \in f(x_0)$ and $p \in f(x_1)$. Then $p \in f(x_0 \sqcap x_1)$, which can only mean $x_0 = x_1$ by minimality. Hence, there is a least threshold for every *p* in $f(x)$.

The proof that $p \in f(\pi_1^{\Rightarrow}(\mathsf{Path}_f(x,p)))$ relies on further invariants of the construction of the paths $p_1, \ldots, p_m$:

1. $\|\pi_1^{\Rightarrow}(p_i)\| \sqsubseteq x$ for all *i*;
2. $\pi_2^{\Rightarrow}(p_i) \sqsubseteq p$ for all *i*;

3. if $\pi_2^{\Rightarrow}(p_i) \in Res_{\mathbf{M}_2}$ then $\|\pi_1^{\Rightarrow}(p_i)\|$ is the minimal element $x' \sqsubseteq x$ such that $\pi_2^{\Rightarrow}(p_i) \in f(x')$;

4. if $\pi_2^{\Rightarrow}(p_i) \in Que_{\mathbf{M}_2}$ then $p_i = q_i \cdot \langle q, 1\rangle$ and $q = \mathsf{si}_f(\|\pi_1^{\Rightarrow}(q_i)\|, \pi_2^{\Rightarrow}(q_i))$.

The first two invariants are obvious. The third and fourth follow from simple arguments: First, if $a_1 d_1 \notin f(\perp)$, then $a_1 \in Open(f(\perp))$. Since $a_1 d_1 \in f(x)$ and $\perp \sqsubseteq x$, $\mathsf{si}_f(\perp, a_1)$ is defined. If $a_1 d_1 \in f(\perp)$ then $\perp$ is clearly the minimal element below *x* such that this is the case. Second, if p_i was constructed via clause 2 of Definition 6.16, and $\pi_2^{\Rightarrow}(p_i) \cdot d_j \notin f(\|\pi_1^{\Rightarrow}(q_{i+1})\|)$ then $\pi_2^{\Rightarrow}(p_i) \in Open(f(\|\pi_1^{\Rightarrow}(q_{i+1})\|))$. Since $\pi_2^{\Rightarrow}(p_i) \cdot d_j \in f(x)$ and $\|\pi_1^{\Rightarrow}(q_{i+1})\| \sqsubseteq x$, $\mathsf{si}_f(\|\pi_1^{\Rightarrow}(q_{i+1})\|, \pi_2^{\Rightarrow}(q_{i+1}))$ is defined. But, if $\pi_2^{\Rightarrow}(p_i) \cdot d_j \in f(\|\pi_1^{\Rightarrow}(q_{i+1})\|)$ then $\|\pi_1^{\Rightarrow}(q_{i+1})\|$ is a minimal element below x such that this is the case. The argument below shows that there is only one minimal element. Finally, if p_i was constructed via clause 3, the arguments proceeds as for clause 2.

Now it suffices to show that p_m outputs p. The lack of a successor for p_m is due to one of two reasons:

1. $p_m = q_m \cdot \langle q, 1\rangle$ and $q \cdot \mathsf{e} \in x$ (it is impossible that $q \in Open(x)$). But then $\pi_2^{\Rightarrow}(q_m) \cdot \mathsf{e} \in f(\|\pi_1^{\Rightarrow}(q_m)\|)$, $\pi_2^{\Rightarrow}(q_m) \cdot \mathsf{e} \in f(x)$, and hence, $p = \pi_2^{\Rightarrow}(q_m) \cdot \mathsf{e}$.

2. $p_m = q_m \cdot d_j$ and $j \not< n$: immediately, $\pi_2^{\Rightarrow}(p_m) = p$.

In either case, $p \in \mathsf{Path}_f(x, p) \star x$. ∎

Two distinct paths $\mathsf{Path}_f(x, p)$ and $\mathsf{Path}_f(x', p')$ cannot interfere with each other. Thus, the collection of all such finite elements forms a valid tree. Moreover, the tree "implements" f when interpreted as a "functional tree".

Lemma 6.18 *1.* $\mathsf{Tree} : \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2) \longrightarrow \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$.

2. If $f \in \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$*, let* $x \in \mathbb{D}(\mathbf{M}_1)$*, then* $f(x) = \mathsf{Tree}(f) \star x$*.*

Proof. Part 1 requires a proof that $\mathsf{Tree}(f)$ is not only prefix-closed, as shown in the preceding lemma, but is also glb-closed. It suffices to show that two finite elements $\mathsf{Path}_f(x_1, p_1)$ and $\mathsf{Path}_f(x_2, p_2)$ in the tree cannot contain paths whose glb is a query. Thus, assume that there is a query *q* that has two incompatible completions to responses in $\mathsf{Path}_f(x_1, p_1)$ and $\mathsf{Path}_f(x_2, p_2)$. There are nine possible cases, each of which falls into one of three classes with $d_1 \neq d_2, q_1 \neq q_2, \mathsf{e}_1 \neq \mathsf{e}_2$:

	$q \cdot \mathsf{e}_2$	$q \cdot \langle d_2, 2\rangle$	$q \cdot \langle q_2, 1\rangle$
$q \cdot \mathsf{e}_1$	A	A	B
$q \cdot \langle d_1, 2\rangle$	A	A	B
$q \cdot \langle q_1, 1\rangle$	B	B	C

We concentrate on one example from each class:

A: Let $q \cdot \langle d_1, 2\rangle \in \mathsf{Path}_f(x_1, p_1)$ and $q \cdot \langle d_2, 2\rangle \in \mathsf{Path}_f(x_2, p_2)$. By Lemma 6.17,

$$\begin{array}{rcl} \pi_2^{\Rightarrow}(q \cdot \langle d_1, 2\rangle) & \in & f(\|\pi_1^{\Rightarrow}(q \cdot \langle d_1, 2\rangle)\|) \\ \pi_2^{\Rightarrow}(q \cdot \langle d_2, 2\rangle) & \in & f(\|\pi_1^{\Rightarrow}(q \cdot \langle d_2, 2\rangle)\|). \end{array}$$

But by the definition of the projection functions,

$$f(\|\pi_1^{\Rightarrow}(q \cdot \langle d_1, 2\rangle)\|) = f(\|\pi_1^{\Rightarrow}(q)\|) = f(\|\pi_1^{\Rightarrow}(q \cdot \langle d_2, 2\rangle)\|) \,.$$

Therefore,

$$\pi_2^{\Rightarrow}(q) \cdot d_1 \in f(\|\pi_1^{\Rightarrow}(q)\|)$$

and

$$\pi_2^{\Rightarrow}(q) \cdot d_2 \in f(\|\pi_1^{\Rightarrow}(q)\|) \,.$$

Hence, $\pi_2^{\Rightarrow}(q)$, a query, is in $f(\|\pi_1^{\Rightarrow}(q \cdot \langle d_2, 2\rangle)\|)$, which proves that the latter is not a tree, contradicting the basic assumptions of the lemma.

B: Let $q \cdot \langle d_1, 2\rangle \in \mathsf{Path}_f(x_1, p_1)$ and $q \cdot \langle q_2, 1\rangle \in \mathsf{Path}_f(x_2, p_2)$. Then

$$\begin{array}{rcl} \pi_2^{\Rightarrow}(q) \cdot d_1 \in f(\|\pi_1^{\Rightarrow}(q \cdot \langle d_1, 2\rangle)\|) & = & f(\|\pi_1^{\Rightarrow}(q)\|), \\ q_2 & = & \mathsf{si}_f(\|\pi_1^{\Rightarrow}(q)\|, \pi_2^{\Rightarrow}(q)). \end{array}$$

But the two consequences simultaneously demand that

$$\pi_2^{\Rightarrow}(q) \cdot d_1 \in \mathit{Answered}(f(\|\pi_1^{\Rightarrow}(q)\|))$$

and

$$\pi_2^{\Rightarrow}(q) \in \mathit{Open}(f(\|\pi_1^{\Rightarrow}(q \cdot \langle q_2, 1\rangle)\|)) = \mathit{Open}(f(\|\pi_1^{\Rightarrow}(q)\|)),$$

that is, we have a contradiction.

C: Let $q \cdot \langle q_1, 1\rangle \in \mathsf{Path}_f(x_1, p_1)$ and $q \cdot \langle q_2, 1\rangle \in \mathsf{Path}_f(x_2, p_2)$. Since si_f is a function for manifestly sequential functions

$$q_1 = \mathsf{si}_f(\|\pi_1^{\Rightarrow}(q)\|, \pi_2^{\Rightarrow}(q)) = q_2,$$

which contradicts the assumption.

The other cases in the three classes have similar proofs. In summary, a case analysis shows that the intersection of two responses from distinct finite elements $\mathsf{Path}_f(x_1, p_1)$ and $\mathsf{Path}_f(x_2, p_2)$ is a response, and that therefore, $\mathsf{Tree}(f) \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$.

To prove part 2, we must show $f(x) \subseteq \mathsf{Tree}(f) \star x$ and $\mathsf{Tree}(f) \star x \subseteq f(x)$, and vice versa. The proof of the first proposition is easy. Assume $p \in f(x)$. By continuity, there exists a finite $x_0 \sqsubseteq x$ such that $p \in f(x_0)$. By Lemma 6.17, $p \in \mathsf{Path}_f(x_0, p) \star x_0$ and, hence, $p \in \mathsf{Tree}(f) \star x_0$. By monotonicity, $p \in \mathsf{Tree}(f) \star x$.

To prove $\mathsf{Tree}(f) \star x \subseteq f(x)$, we must consider two cases. Let $p \in \mathsf{Tree}(f) \star x$. In the first case, $r \in \mathsf{Tree}(f)$, $\pi_2^{\Rightarrow}(r) = p$, and $\|\pi_1^{\Rightarrow}(r)\| \sqsubseteq x$. By the construction of paths in $\mathsf{Tree}(f)$,

$$p = \pi_2^{\Rightarrow}(r) \in f(\|\pi_1^{\Rightarrow}(r)\|).$$

By monotonicity, $p \in f(x)$. In the second case, $q \cdot \langle q', 1\rangle \in \mathsf{Tree}(f), \pi_2^{\Rightarrow}(q) \cdot \mathsf{e} = p$, and $\|\pi_1^{\Rightarrow}(q)\| \cup \{q' \cdot \mathsf{e}\} \sqsubseteq x$. Now the construction of $q \cdot \langle q', 1\rangle$ implies that

$$q' = \mathsf{si}_f(\|\pi_1^{\Rightarrow}(q)\|, \pi_2^{\Rightarrow}(q)),$$

which means $\pi_2^{\Rightarrow}(q) \in Open(f(x))$. But then

$$p = \pi_2^{\Rightarrow}(q) \cdot \mathsf{e} \in f(\|\pi_1^{\Rightarrow}(q)\| \cup \{q' \cdot \mathsf{e}\})$$

by error propagation. The conclusion, $p \in f(x)$, follows again by monotonicity. Together the two cases imply $\mathsf{Tree}(f) \star x = f(x)$. ∎

In summary, we have shown that there exists a bijection between the set $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ of manifestly sequential functions and the (carrier of the) domain over the exponent *sds* $\mathbf{M}_1 \Rightarrow \mathbf{M}_2$.

Theorem 6.19 *1. For every manifestly sequential function $f \in \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$,*

$$\mathsf{Fun}(\mathsf{Tree}(f)) = f.$$

2. For every domain element $t \in \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$,

$$\mathsf{Tree}(\mathsf{Fun}(t)) = t.$$

Proof. To prove part 1, we observe that for all x,

$$\mathsf{Fun}(\mathsf{Tree}(f))(x) = \mathsf{Tree}(f) \star x = f(x)$$

by Lemma 6.18. By extensionality, $\mathsf{Fun}(\mathsf{Tree}(f)) = f$.

To prove part 2, we observe that for all x,

$$\mathsf{Tree}(\mathsf{Fun}(t)) \star x = \mathsf{Fun}(t)(x) = t \star x$$

by Lemma 6.18 and Definition 6.14. By the Extensionality Theorem (6.12), $\mathsf{Tree}(\mathsf{Fun}(t)) = t$. ∎

Together with the third part of the Extensionality Theorem, the theorem implies that the function space $\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ itself with the usual, pointwise ordering is a Scott domain provided the error set contains at least two elements.

Notation We write $f \sqsubseteq_e g$ iff for all x, $f(x) \sqsubseteq g(x)$.

Corollary 6.20 *If $\mathbb{E}$ contains at least two elements, then* $\mathsf{Fun} : \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2) \longrightarrow \mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ *is an order-isomorphism. Hence, $(\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2), \sqsubseteq_e)$ is a Scott domain.*[16]

Proof. By Theorem 6.19, Fun is a bijection. Under the additional assumption of the corollary, $t \sqsubseteq t'$ iff $t \star x \sqsubseteq t' \star x$ for all x (by Extensionality) iff $\mathsf{Fun}(t)(x) \sqsubseteq \mathsf{Fun}(t')(x)$ for all x (by Lemma 6.18). ∎

[16] Interestingly the Berry-Curien stable ordering for this space coincides with the pointwise ordering [11].

6.2.2 The Exponent Object

The fact that the decision tree representation for manifestly sequential functions is extensional and that the tree domain is isomorphic to the function space of manifestly sequential functions provides the background for the definition of the exponent object and its related arrows, Λ and Λ^{-1}. The following definition is "operational" in that it defines currying and uncurrying as renaming operations for paths in the trees.

Definition 6.21 (*Exponent Object,* Λ**)** Let $\mathbf{M}_i$ be *sds*'s, and let $\mathbb{D}(\mathbf{M}_i)$ be their respective domains for $i \in \{0,1,2\}$. $\mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ is the exponent object for $\mathbb{D}(\mathbf{M}_1)$ and $\mathbb{D}(\mathbf{M}_2)$. For arbitrary $\mathbb{D}(\mathbf{M}_2)$, $\Lambda : [\mathbb{D}(\mathbf{M}_0 \times \mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)] \longrightarrow [\mathbb{D}(\mathbf{M}_0) \longrightarrow \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)]$ and its inverse Λ^{-1} are defined as follows:

$$
\begin{array}{rcl}
\Lambda(f) & = & \mathsf{Fun}(\Lambda_t(\mathsf{Tree}(f))) \\
& & \text{where } \Lambda_t : \mathbb{D}((\mathbf{M}_0 \times \mathbf{M}_1) \Rightarrow \mathbf{M}_2) \longrightarrow \mathbb{D}(\mathbf{M}_0 \Rightarrow (\mathbf{M}_1 \Rightarrow \mathbf{M}_2)) \\
& & \qquad \Lambda_t(x) = \{\Lambda_t(p) \mid p \in x\} \\
& & \text{and } \quad \begin{array}{lcl}
\Lambda_t(p \cdot \langle p_0, 1\rangle) & = & \Lambda_t(p) \cdot \langle \pi_1^{\times}(p_0), 1\rangle \\
& & \text{if for some } a \in A_0, p_0 = \langle a, 1\rangle \ldots \\
\Lambda_t(p \cdot \langle p_1, 1\rangle) & = & \Lambda_t(p) \cdot \langle\langle \pi_2^{\times}(p_1), 1\rangle, 2\rangle \\
& & \text{if for some } a \in A_1, p_1 = \langle a, 2\rangle \ldots \\
\Lambda_t(p \cdot \langle y, 2\rangle) & = & \Lambda_t(p) \cdot \langle\langle y, 2\rangle, 2\rangle \\
\Lambda_t(p \cdot \mathsf{e}) & = & \Lambda_t(p) \cdot \mathsf{e}
\end{array} \\
\Lambda^{-1}(g) & = & \mathsf{Fun}(\Lambda_t^{-1}(\mathsf{Tree}(g))) \\
& & \text{where } \Lambda_t^{-1} : \mathbb{D}(\mathbf{M}_0 \Rightarrow (\mathbf{M}_1 \Rightarrow \mathbf{M}_2)) \longrightarrow \mathbb{D}((\mathbf{M}_0 \times \mathbf{M}_1) \Rightarrow \mathbf{M}_2) \\
& & \qquad \Lambda_t^{-1}(x) = \{\Lambda_t^{-1}(p) \mid p \in x\} \\
& & \text{and } \quad \begin{array}{lcl}
\Lambda_t^{-1}(p \cdot \langle p_0, 1\rangle) & = & \Lambda_t^{-1}(p) \cdot \langle inj_1(p_0), 1\rangle \\
\Lambda_t^{-1}(p \cdot \langle\langle p_1, 1\rangle, 2\rangle) & = & \Lambda_t^{-1}(p) \cdot \langle inj_2(p_1), 1\rangle \\
\Lambda_t^{-1}(p \cdot \langle\langle y, 2\rangle, 2\rangle) & = & \Lambda_t^{-1}(p) \cdot \langle y, 2\rangle \\
\Lambda_t^{-1}(p \cdot \mathsf{e}) & = & \Lambda_t^{-1}(p) \cdot \mathsf{e}
\end{array}
\end{array}
$$

∎

To verify that the definition is correct, it suffices to check the appropriate equations with simple calculations.

Lemma 6.22 *Let* $f \in \mathbb{D}(\mathbf{M}_0 \times \mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$ *and let* $g \in \mathbb{D}(\mathbf{M}_0) \longrightarrow \mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$. *Then* $\Lambda(f)$ *and* $\Lambda^{-1}(g)$ *are manifestly sequential functions, and*

1. $\Lambda^{-1}(\Lambda(f)) = f$;
2. $\Lambda(\Lambda^{-1}(g)) = g$;
3. $\Lambda(f)(x_0) \star x_1 = f(inj_1(x_0) \cup inj_2(x_1))$ *for all* $x_0 \in \mathbb{D}(\mathbf{M}_0)$ *and* $x_1 \in \mathbb{D}(\mathbf{M}_1)$*; and*
4. $\Lambda(f) \circ g = \Lambda(f \circ \langle g \circ \pi_1, \pi_2\rangle)$, *for any g of correct type.*

Proof. The validity of claims 1 and 2 clearly follows from the validity of analogous claims about $\Lambda_t(\mathsf{Tree}(f))$ and $\Lambda_t^{-1}(\mathsf{Tree}(g))$ due to Theorem 6.19, which, in turn, follow from the respective claims for *paths*. Finally, Λ_t and Λ_t^{-1} are obviously bijections for paths, *i.e.*, for all paths p over $(\mathbf{M}_0 \times \mathbf{M}_1) \Rightarrow \mathbf{M}_2$, $\Lambda_t^{-1}(\Lambda_t(p)) = p$; and for all paths p over $\mathbf{M}_0 \Rightarrow (\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$, $\Lambda_t(\Lambda_t^{-1}(p)) = p$. The proof is by induction on the length of p.

The proof of claim 3 requires the verification of the following properties of a response p over $(\mathbf{M}_0 \times \mathbf{M}_1) \Rightarrow \mathbf{M}_2$ and its curried version $\Lambda_t(p)$:

$$\begin{array}{rcl}
\pi_1^{\times}(\|\pi_1^{\Rightarrow}(p)\|) & = & \|\pi_1^{\Rightarrow}(\Lambda_t(p))\| \\
\pi_2^{\times}(\|\pi_1^{\Rightarrow}(p)\|) & = & \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(p)))\| \\
\pi_2^{\Rightarrow}(p) & = & \pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(p)))
\end{array}$$

Again, a proof by an induction on the length of the response p verifies the three claims. The rest follows easily:

$$\begin{array}{rl}
& \Lambda(f)(x_0) \star x_1 \\
= & \mathsf{Fun}(\Lambda_t(\mathsf{Tree}(f)))(x_0) \star x_1 \\
= & \Lambda_t(\mathsf{Tree}(f)) \star x_0 \star x_1 \\
= & (\{\pi_2^{\Rightarrow}(r) \,|\, r \in \Lambda_t(\mathsf{Tree}(f)), \|\pi_1^{\Rightarrow}(r)\| \sqsubseteq x_0\} \\
& \cup \{\pi_2^{\Rightarrow}(p) \cdot \mathsf{e} \,|\, p \cdot \langle q, 1\rangle \in \Lambda_t(\mathsf{Tree}(f)), \|\pi_1^{\Rightarrow}(p)\| \cup \{q \cdot \mathsf{e}\} \sqsubseteq x_0\}) \star x_1 \\
= & \{\pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(r)) \,|\, r \in \Lambda_t(\mathsf{Tree}(f)), \|\pi_1^{\Rightarrow}(r)\| \sqsubseteq x_0, \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(r))\| \sqsubseteq x_1\} \\
& \cup \{\pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(p)) \cdot \mathsf{e} \,| \\
& \quad p \cdot \langle\langle q, 1\rangle, 2\rangle \in \Lambda_t(\mathsf{Tree}(f)), \|\pi_1^{\Rightarrow}(p)\| \sqsubseteq x_0, \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(p))\| \cup \{q \cdot \mathsf{e}\} \sqsubseteq x_1\} \\
& \cup \{\pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(p)) \cdot \mathsf{e} \,| \\
& \quad p \cdot \langle q, 1\rangle \in \Lambda_t(\mathsf{Tree}(f)), \|\pi_1^{\Rightarrow}(p)\| \cup \{q \cdot \mathsf{e}\} \sqsubseteq x_0, \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(p))\| \sqsubseteq x_1\} \\
& \\
= & \{\pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(r))) \,|\, r \in \mathsf{Tree}(f), \|\pi_1^{\Rightarrow}(\Lambda_t(r))\| \sqsubseteq x_0, \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(r)))\| \sqsubseteq x_1\} \\
& \cup \{\pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(p))) \cdot \mathsf{e} \,| \\
& \quad p \cdot \langle q, 1\rangle \in \mathsf{Tree}(f), \|\pi_1^{\Rightarrow}(\Lambda_t(p))\| \sqsubseteq x_0, \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(p)))\| \cup \{\pi_2^{\times}(q) \cdot \mathsf{e}\} \sqsubseteq x_1\} \\
& \cup \{\pi_2^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(p))) \cdot \mathsf{e} \,| \\
& \quad p \cdot \langle q, 1\rangle \in \mathsf{Tree}(f), \|\pi_1^{\Rightarrow}(\Lambda_t(p))\| \cup \{\pi_1^{\times}(q) \cdot \mathsf{e}\} \sqsubseteq x_0, \|\pi_1^{\Rightarrow}(\pi_2^{\Rightarrow}(\Lambda_t(p)))\| \sqsubseteq x_1\} \\
& \\
= & \{\pi_2^{\Rightarrow}(r) \,|\, r \in \mathsf{Tree}(f), \pi_1^{\times}(\|\pi_1^{\Rightarrow}(r)\|) \sqsubseteq x_0, \pi_2^{\times}(\|\pi_1^{\Rightarrow}(r)\|) \sqsubseteq x_1\} \\
& \cup \{\pi_2^{\Rightarrow}(p) \cdot \mathsf{e} \,| \\
& \quad p \cdot \langle q, 1\rangle \in \mathsf{Tree}(f), \pi_1^{\times}(\|\pi_1^{\Rightarrow}(p)\|) \sqsubseteq x_0, \pi_2^{\times}(\|\pi_1^{\Rightarrow}(p)\|) \cup \{\pi_2^{\times}(q) \cdot \mathsf{e}\} \sqsubseteq x_1\} \\
& \cup \{\pi_2^{\Rightarrow}(p) \cdot \mathsf{e} \,| \\
& \quad p \cdot \langle q, 1\rangle \in \mathsf{Tree}(f), \pi_1^{\times}(\|\pi_1^{\Rightarrow}(p)\|) \cup \{\pi_1^{\times}(q) \cdot \mathsf{e}\} \sqsubseteq x_0, \pi_2^{\times}(\|\pi_1^{\Rightarrow}(p)\|) \sqsubseteq x_1\} \\
& \\
= & \{\pi_2^{\Rightarrow}(p) \,|\, p \in \mathsf{Tree}(f), \|\pi_1^{\Rightarrow}(r)\| \sqsubseteq inj_1(x_0) \cup inj_2(x_1)\} \\
& \cup \{\pi_2^{\Rightarrow}(p) \cdot \mathsf{e} \,|\, p \cdot \langle q, 1\rangle \in \mathsf{Tree}(f), \|\pi_1^{\Rightarrow}(p)\| \cup \{q \cdot \mathsf{e}\} \sqsubseteq inj_1(x_0) \cup inj_2(x_1)\}
\end{array}$$

$$
\begin{aligned}
&= \mathsf{Tree}(f) \star (inj_1(x_0) \cup inj_2(x_1)) \\
&= f(inj_1(x_0) \cup inj_2(x_1))
\end{aligned}
$$

By claim 3, currying and uncurrying as defined are just the set-theoretic currying and uncurrying functions, hence the equation of claim 4 holds in $\mathcal{SEQ}$ since it holds in the category of sets. ∎

7 Full Abstraction for SPCF

Both $\mathcal{SEQ}(\mathbb{E})$ and $\mathcal{S}eq(\mathbb{E})$ can serve as the basis of a fully abstract semantics for SPCF. Unfortunately, working with the elements in the domains of either category imposes a heavy notational overhead. To overcome this problem, we introduce some additional notation.

Consider the meaning of add1 in $\mathcal{SEQ}(\mathbb{E})$. It is the arrow $f : 1 \longrightarrow \mathbb{D}(\mathbf{N} \Rightarrow \mathbf{N})$, and in $\mathcal{S}eq(\mathbb{E})$ it is the arrow $f' : 1 \longrightarrow \mathbb{D}(\mathbf{N} \Rightarrow \mathbf{N})$ where

$$f' \star x = f(x) = A = \{\langle ?, 2\rangle \cdot \langle ?, 1\rangle, \langle ?, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ?n, 1\rangle \cdot \langle ?m, 2\rangle \mid m, n \in \mathbb{N}, m = n+1\}$$

for the unique element x in 1. Similarly, the procedure if0 is the arrow that produces the following element

$$
\left\{ \begin{array}{l}
\langle\langle\langle ?, 2\rangle, 2\rangle, 2\rangle \cdot \langle ?, 1\rangle, \\
\langle\langle\langle ?, 2\rangle, 2\rangle, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot 0, 1\rangle \cdot \langle\langle ?, 1\rangle, 2\rangle, \\
\langle\langle\langle ?, 2\rangle, 2\rangle, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot 0, 1\rangle \cdot \langle\langle ?, 1\rangle, 2\rangle \cdot \langle\langle ? \cdot m, 1\rangle, 2\rangle \cdot \langle\langle\langle m, 2\rangle, 2\rangle, 2\rangle, \\
\langle\langle\langle ?, 2\rangle, 2\rangle, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot n, 1\rangle \cdot \langle\langle ?, 1\rangle, 2\rangle, \\
\langle\langle\langle ?, 2\rangle, 2\rangle, 2\rangle \cdot \langle ?, 1\rangle \cdot \langle ? \cdot n, 1\rangle \cdot \langle\langle ?, 1\rangle, 2\rangle \cdot \langle\langle ? \cdot m, 1\rangle, 2\rangle \cdot \langle\langle\langle m, 2\rangle, 2\rangle, 2\rangle
\end{array} \;\middle|\; n > 0 \right\}
$$

in $\mathbb{D}(\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N})$ upon application to the single element of 1. To simplify the definition of SPCF's semantics and the proofs about the semantics, we first introduce some notational abbreviations before we specify some trees that are common to both semantics.

Since all SPCF types τ have the shape

$$\tau = \tau_1 \to \ldots \to \tau_k \to o, \text{ for } k \geq 0,$$

the interesting domains are generated by *sds*'s of the shape

$$\mathbf{M} = \mathbf{M}_1 \Rightarrow \ldots \mathbf{M}_k \Rightarrow \mathbf{N}$$

where $\mathbf{M}_i$ is the *sds* that corresponds to τ_i and $\mathbf{N}$ is the *sds* that generates $\mathbb{N}^e_\perp$, the flat domain of natural numbers and errors (cmp. the examples following Definition 5.1). By the definition of the exponent construction, all paths over $\mathbb{D}(\mathbf{M})$ start with the initial address

$$\langle \ldots \langle ?, \underbrace{2\rangle, \ldots, 2\rangle}_{k}.$$

$$\mathbb{D} = \mathbb{D}(\mathbf{M}_1 \Rightarrow \ldots \mathbf{M}_k \Rightarrow \mathbf{N}) \text{ where } \mathbf{M}_i = \mathbf{M}_{i,1} \Rightarrow \ldots \mathbf{M}_{i,k_i} \Rightarrow \mathbf{N}$$

$$\begin{array}{rcll}
\langle ? \rangle & = & \langle \ldots \langle ?, \underbrace{2 \rangle, \ldots, 2 \rangle}_{k} & \\
\langle n \rangle & = & \langle \ldots \langle ? \cdot n, \underbrace{2 \rangle, \ldots, 2 \rangle}_{k} & \\
\langle p, i \rangle & = & \langle \ldots \langle p, 1 \rangle, \underbrace{2 \rangle, \ldots, 2 \rangle}_{i-1} & \qquad \pi_i(p) = \pi_1^{\Rightarrow}(\underbrace{\pi_2^{\Rightarrow}(\ldots \pi_2^{\Rightarrow}}_{i-1}(p) \ldots) \\
\langle ?, i \rangle & = & \langle \langle ? \rangle, i \rangle &
\end{array}$$

Figure 5: Abbreviations for domain elements in an SPCF domain

Below we use $\langle ? \rangle$ as an abbreviation for this address, unless $k = 0$. Similarly, a maximal path in $\mathbb{D}(\mathbf{M})$ ends in $\mathsf{e} \in \mathbb{E}$ or in

$$\langle \ldots \langle n, \underbrace{2 \rangle, \ldots, 2 \rangle}_{k},$$

for $n \in \mathbb{N}$. We will use $\langle n \rangle$ to denote this datum. Finally, the intermediate queries and responses of a path in $\mathbb{D}(\mathbf{M})$ are about elements in $\mathbb{D}(\mathbf{M}_1)$ through $\mathbb{D}(\mathbf{M}_k)$ and, for domain $\mathbb{D}(\mathbf{M}_i)$ have the shape

$$\langle \ldots \langle p, 1 \rangle, \underbrace{2 \rangle, \ldots, 2 \rangle}_{i-1},$$

for $1 \leq i \leq k$. The abbreviation $\langle p, i \rangle$ will stand for the above. For the very first query about the *i*th argument, we use $\langle ?, i \rangle$ instead of $\langle \langle ? \rangle, i \rangle$. To extract the path *p* from $\langle p, i \rangle$ we use the function π_i, which is the composition of $\pi_1^{\Rightarrow}$ with the $(i-1)$-fold composition of $\pi_2^{\Rightarrow}$.

Warning: When encountering the abbreviations below, the reader should keep in mind that the implicit parameter *k* is a context-sensitive entity.

Using the newly introduced abbreviations, we turn to the definition of trees that characterize the denotations of `add1`, `sub1`, `if0`, and `catch`.

Definition 7.1 (*A*, *S*, *I*, *C***)** Let $\mathbf{M}_1, \ldots, \mathbf{M}_k$ be arbitrary *sds*'s. Then the constants

$$\begin{array}{rcl}
A, S & \in & \mathbb{D}(\mathbf{N} \Rightarrow \mathbf{N}) \\
I & \in & \mathbb{D}(\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}) \\
C_k & \in & \mathbb{D}((\mathbf{M}_1 \Rightarrow \ldots \Rightarrow \mathbf{M}_k \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N})
\end{array}$$

are defined as follows:

$$A \;=\; \{\, \langle ? \rangle \cdot \langle ?, 1 \rangle,\ \langle ? \rangle \cdot \langle ?, 1 \rangle \cdot \langle ?n, 1 \rangle \cdot \langle m \rangle \mid m, n \in \mathbb{N}, m = n + 1 \,\}$$

$$S = \{ \langle ?\rangle \cdot \langle ?, 1\rangle,\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle ?n, 1\rangle \cdot \langle m\rangle \mid m, n \in \mathbb{N}, m+1 = n \}$$

$$I = \left\{ \begin{array}{l} \langle ?\rangle \cdot \langle ?, 1\rangle, \\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle ?\cdot 0, 1\rangle \cdot \langle ?, 2\rangle, \\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle ?\cdot 0, 1\rangle \cdot \langle ?, 2\rangle \cdot \langle ?\cdot m, 2\rangle \cdot \langle m\rangle, \\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle ?\cdot n, 1\rangle \cdot \langle ?, 3\rangle, \\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle ?\cdot n, 1\rangle \cdot \langle ?, 3\rangle \cdot \langle ?\cdot m, 3\rangle \cdot \langle m\rangle \end{array} \middle| \; n, m \in \mathbb{N}, n > 0 \right\}$$

$$C_k = \left\{ \begin{array}{l} \langle ?\rangle \cdot \langle ?, 1\rangle, \\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle \langle ?\rangle \cdot \langle ?, i\rangle, 1\rangle \cdot \langle i-1\rangle, \\ \langle ?\rangle \cdot \langle ?, 1\rangle \cdot \langle \langle n\rangle, 1\rangle \cdot \langle k+n\rangle \end{array} \middle| \; n, i \in \mathbb{N}, 1 \le i \le k \right\}$$

∎

While the definitions for A and S are obvious, those for I and C_k deserve some additional explanation. I represents a function that contains the following evaluation strategy. First, it announces that it will fill the only address $?$ in the output space and then probes its first argument. If it is 0, it probes the second argument and puts its answer at the end of the output path; if not, it probes the third argument and continues as before. As for C_k, it also probes its first argument. If the argument is a constant function that would return n, C_k returns $k+n$. If the argument represents a function that is strict in its ith argument, then C_k returns $i-1$.

We are now ready to define the semantics of SPCF.

Definition 7.2 (*Seq and $\mathcal{SEQ}$ Semantics of SPCF***)** Let $\mathbb{E}$ be a set of error values.

If $\mathbb{E} \neq \emptyset$, the underlying cartesian-closed category of SPCF's $\mathcal{SEQ}$ semantics is $\mathcal{SEQ}(\mathbb{E})$. The ground type o is interpreted as the domain $\mathbb{N}^e_\perp$:

$$\mathcal{C}^o = \mathbb{D}(\mathbf{N}).$$

The $\mathcal{SEQ}$ semantics assigns the following meaning to constants:

$$\begin{array}{rcll} \mathcal{SEQ}[\![\, \mathsf{e} \,]\!] & = & \underline{\lambda}\rho : 1.\mathsf{e} & \text{for } \mathsf{e} \in \mathbb{E} \\ \mathcal{SEQ}[\![\, \ulcorner n \urcorner \,]\!] & = & \underline{\lambda}\rho : 1.n & \\ \mathcal{SEQ}[\![\, \mathsf{add1} \,]\!] & = & \underline{\lambda}\rho : 1.A & \\ \mathcal{SEQ}[\![\, \mathsf{sub1} \,]\!] & = & \underline{\lambda}\rho : 1.S & \\ \mathcal{SEQ}[\![\, \mathsf{if0} \,]\!] & = & \underline{\lambda}\rho : 1.I & \\ \mathcal{SEQ}[\![\, \mathsf{catch}_k \,]\!] & = & \underline{\lambda}\rho : 1.C_k & \end{array}$$

The meaning of terms in the $\mathcal{SEQ}$ semantics are functions that map n-tuples or *environments* to elements in the appropriate domain. To avoid notational clutter, we use tuple notation for environments: if M's free variables are among $x_1^{\tau_1} \ldots x_k^{\tau_k}$, we write

$$\mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash M \,]\!]\langle x_1, \ldots, x_k\rangle$$

for

$$\mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash M \,]\!]\rho$$

where $x_i = \pi_1^\times(\underbrace{\pi_2^\times(\ldots\pi_2^\times}_{i-1}(\rho)\ldots))$ for arbitrary $x_i \in \mathcal{SEQ}^{\tau_i}$.

For arbitrary $\mathbb{E}$, the underlying cartesian-closed category of SPCF's *Seq* semantics is *Seq*($\mathbb{E}$). The ground type o is interpreted as $\mathbb{N}_\perp^e$:

$$\mathcal{C}^o = \mathbb{D}(\mathbf{N}).$$

The *Seq* semantics assigns the following meaning to constants:

$$\begin{array}{rcl}
\mathit{Seq}[\![\, \mathsf{e} \,]\!] & = & \{\langle ?, 2\rangle \cdot \mathsf{e}\} \\
\mathit{Seq}[\![\, \ulcorner n \urcorner \,]\!] & = & \{\langle ?, 2\rangle \cdot \langle n, 2\rangle\}) \\
\mathit{Seq}[\![\, \mathsf{add1} \,]\!] & = & \{\langle b_1, 2\rangle \cdot \ldots \cdot \langle b_n, 2\rangle \mid b_1 \cdot \ldots \cdot b_n \in A\} \\
\mathit{Seq}[\![\, \mathsf{sub1} \,]\!] & = & \{\langle b_1, 2\rangle \cdot \ldots \cdot \langle b_n, 2\rangle \mid b_1 \cdot \ldots \cdot b_n \in S\} \\
\mathit{Seq}[\![\, \mathsf{if0} \,]\!] & = & \{\langle b_1, 2\rangle \cdot \ldots \cdot \langle b_n, 2\rangle \mid b_1 \cdot \ldots \cdot b_n \in I\} \\
\mathit{Seq}[\![\, \mathsf{catch}_k \,]\!] & = & \{\langle b_1, 2\rangle \cdot \ldots \cdot \langle b_n, 2\rangle \mid b_1 \cdot \ldots \cdot b_n \in I\}
\end{array}$$

The meaning of terms in the *Seq* semantics are algorithms that input *n*-tuples and output elements in the appropriate domain. Again, we use ordinary tuple notation for these environments.

We use $\mathit{Seq}^{\mathbb{E}}[\![\ \cdot\ ]\!]$ when we need to distinguish between two *Seq* semantics based on different error sets. ∎

For $\mathbb{E} \neq \emptyset$, we interchangeably use the $\mathcal{SEQ}$ and *Seq* semantics for SPCF since

$$\mathsf{Fun}(\mathit{Seq}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash M \,]\!]) = \mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash M \,]\!]$$

for all SPCF terms M whose free variables are among $x_1^{\tau_1} \ldots x_k^{\tau_k}$. Similarly, for a term that does not explicitly mention error, it does not matter which error set we choose to determine its *Seq* semantics.

Lemma 7.3 *For all SPCF($\emptyset$) terms M (that is, a term that does not contain error constants) whose free variables are among $x_1^{\tau_1} \ldots x_k^{\tau_k}$ then for any $\mathbb{E}$,*

$$\mathit{Seq}^{\mathbb{E}}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash M \,]\!] = \mathit{Seq}^{\emptyset}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash M \,]\!].$$

Proof. The proof proceeds by induction on the structure of *M*. ∎

Moreover, SPCF is PCF-like.

Proposition 7.4 *If M is a PCF program, then*

$$\mathcal{D}om[\![\, \emptyset \vdash M \,]\!] \perp = \mathit{Seq}[\![\, \emptyset \vdash M \,]\!]\star \perp = \mathcal{SEQ}[\![\, \emptyset \vdash M \,]\!] \perp .$$

Proof. The theorem follows from the operational adequacy theorem (Theorem 8.13 in the following section), from Plotkin's operational adequacy theorem for the continuous semantics [24:Theorem 3.1], and a simple check that the two versions of the operational semantics agree on PCF. ∎

Next we can show that SPCF is sequential and propagates errors as expected, *i.e.*, that it is an observably sequential programming language. The proof relies on the results in the preceding section and on Lemma 3.6, which states that a semantics based on cartesian-closed categories satisfies (β).

Theorem 7.5 *SPCF is sequential, manifestly sequential (if $\mathbb{E} \neq \emptyset$), and observably sequential.*

Proof. Let $M_1, \ldots, M_k$ be closed phrases in SPCF and let $C[M_1, \ldots, M_k]$ be a program satisfying the hypotheses of Definition 3.9:

(i)**:** $\mathcal{SEQ}[\![\vdash C[M_1, \ldots, M_k]]\!] \in \mathbb{N} \cup \mathbb{E}$;

(ii)**:** $\mathcal{SEQ}[\![\vdash C[\Omega, \ldots, \Omega]]\!] = \perp$.

We must show that there is an argument position j that forces divergence. Since the semantics satisfies the equation β, we have:

$$\begin{array}{rl} & \mathcal{SEQ}[\![\vdash C[M_1, \ldots, M_k]]\!] \perp \\ = & \mathcal{SEQ}[\![\vdash ((\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.C[x_1, \ldots, x_k])\ M_1\ \ldots\ M_k)]\!] \perp \\ = & \mathcal{SEQ}[\![\vdash (\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.C[x_1, \ldots, x_k])]\!] \perp \star \mathcal{SEQ}[\![M_1]\!] \perp \star \ldots \star \mathcal{SEQ}[\![M_k]\!] \perp \end{array}$$

Now consider the possible denotations of the *k*-ary procedure

$$(\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.C[x_1, \ldots, x_k])$$

The denotation of such a procedure is either a decision tree of the shape

$$\{\langle ? \rangle \cdot \langle d \rangle\}$$

for $d \in \mathbb{N} \cup \mathbb{E}$ or a decision tree that contains the path

$$\langle ? \rangle \cdot \langle ?, j \rangle$$

for some $j \leq k$. Clearly, the first case contradicts assumption (ii). The second case specifies that the *k*-ary procedure first probes its *j*th argument. But this fact implies that

$$\begin{array}{rl} & \mathcal{SEQ}[\![\vdash C[M_1', \ldots, M_j', \ldots, M_k']]\!] \\ = & \mathcal{SEQ}[\![\vdash \lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.C[x_1, \ldots, x_k]]\!] \perp \\ & \quad \star \mathcal{SEQ}[\![\vdash M_1']\!] \perp \star \ldots \star \mathcal{SEQ}[\![\vdash M_j']\!] \perp \star \ldots \star \mathcal{SEQ}[\![\vdash M_k']\!] \perp \\ = & \perp \end{array}$$

for $M_j' = \Omega$ and arbitrary M_i', $i \neq j$. More concisely, the program diverges if the expression M_j in *j*th hole diverges—regardless of the expressions in the other holes. Hence, *j* is the desired sequentiality index.

The proof that SPCF programs also propagate errors that occur at the sequentiality index proceeds in precisely the same manner. And finally, given $\lambda x_1 \ldots x_k.C[x_1, \ldots, x_k]$, the expression ($\mathtt{add1}$ ($\mathtt{catch}$ $\lambda x_1 \ldots x_k.C[x_1, \ldots, x_k]$)) produces the sequentiality context, *i.e.*, $D[\ ] =$ ($\mathtt{add1}$ ($\mathtt{catch}$ [ ])) is the appropriate context for any such procedure. Hence, SPCF satisfies the conditions of Definition 4.2, which shows that it is a manifestly sequential and observably sequential language. ∎

After establishing the basic properties of SPCF's semantics, we are ready to prove that the semantics are fully abstract.

Theorem 7.6 (Full Abstraction) *For all error sets* $\mathbb{E}$*,*

1. $M \sqsubseteq_{\mathcal{S}eq} N$ *iff* $M \lesssim N$ *and* $M \equiv_{\mathcal{S}eq} N$ *iff* $M \simeq N$*;*
2. *if* $\mathbb{E} \neq \emptyset$*,* $M \sqsubseteq_{\mathcal{SEQ}} N$ *iff* $M \lesssim N$ *and* $M \equiv_{\mathcal{SEQ}} N$ *iff* $M \simeq N$*.*

Note. We concentrate on the proof of the theorem's second part, indicating differences to the proof of the first part as we proceed.

Proof. The left to right direction follows from the compositional definition of the semantics. For the right to left direction, assume $M \not\sqsubseteq_{\mathcal{SEQ}} N$ (both are closed and of type τ). We will prove that $M \not\lesssim N$.

By the Discriminator Lemma (7.7), there exists a finite, error-free discriminator F of type $\tau \to o$ such that $F \star \mathcal{SEQ}[\![\ \vdash M\ ]\!]\ \bot \not\sqsubseteq\ F \star \mathcal{SEQ}[\![\ \vdash N\ ]\!]\ \bot$. It follows from the Representability Lemma (7.8) that there is a term D such that $\mathcal{SEQ}[\![\ \vdash D\ ]\!] = F$ ($\mathcal{S}eq[\![\ \vdash D\ ]\!] = F$). Thus,

$$\mathcal{SEQ}[\![\ \vdash (D\ M)\ ]\!] \not\sqsubseteq \mathcal{SEQ}[\![\ \vdash (D\ N)\ ]\!],$$

($\mathcal{S}eq[\![\ \vdash (D\ M)\ ]\!] \not\sqsubseteq \mathcal{S}eq[\![\ \vdash (D\ N)\ ]\!]$) which implies $M \not\lesssim M'$ as desired.

For open terms, it suffices to point out that if $x_1^{\tau_1}, \ldots, x_k^{\tau_k}$ are all the free variables in M and N, then $M \lesssim N$ implies $\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.M \lesssim \lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.N$, which by the preceding arguments for closed terms is equivalent to $\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.M \sqsubseteq_{\mathcal{SEQ}} \lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.N$ and hence, $M \sqsubseteq_{\mathcal{SEQ}} N$ [24]. ∎

Lemma 7.7 (Discriminator) *If* f, g *are elements in* $\mathcal{SEQ}^\tau$ *(*$\mathcal{S}eq^\tau$*) and* $f \not\sqsubseteq g$ *then for some finite, error-free* $F \in \mathbb{D}^{\tau \to o}$*,* $F \star f \sqsubseteq F \star g$*. Moreover, by monotonicity,* $f \sqsubseteq g$ *if and only if for all finite* $F \in \mathbb{D}^{\tau \to o}$*,* $F \star f \sqsubseteq F \star g$*.*

Proof. Assume $f \not\sqsubseteq g$. From this it follows that there is some minimal response $r \in f$ such that $r \notin g$: for any $r' \sqsubseteq r$, $r' \in g$. Let $r_1, \ldots, r_n$ be the prefixes of r such that $r_1 \sqsubseteq \ldots \sqsubseteq r_n = r$, let q_i be the query that is the immediate predecessor of r_i, and set

$$F^* = \langle ?, 2\rangle \cdot \langle q_1, 1\rangle \cdot \langle r_1, 1\rangle \cdot \ldots \cdot \langle q_n, 1\rangle \cdot \langle r_n, 1\rangle \cdot \langle 0, 2\rangle.$$

Then if r does not end in an error, F is the prefix-closure of F^*, i.e., $F = \{r \sqsubseteq F^*\}$. As a function, F explores its argument and determines whether the argument contains r. If so, the argument dominates f and F returns 0 ($F \star f = \{? \cdot 0\}$); otherwise it is undefined or propagates errors ($F \star g = \bot$ or $F \star g = \{? \cdot \mathtt{e}\}$).

If $r = q \cdot \mathsf{e}$, the discriminator is the prefix-closure of the shorter path

$$F^{\&} = \langle ?, 2\rangle \cdot \langle q_1, 1\rangle \cdot \langle r_1, 1\rangle \cdot \ldots \cdot \langle q_n, 1\rangle.$$

Again F checks whether $r \in f$ but, if so, it returns e ($F \star f = \{? \cdot \mathsf{e}\}$). If F does not find r, it is undefined or propagates a different error.

Thus, in both cases, $F \star f \not\sqsubseteq F \star g$, which is what the lemma claims. ∎

Lemma 7.8 (Representability) *Let $\tau = \tau_1 \to \ldots \to \tau_k \to o$ for $k \geq 0$. If x is a finite tree in $\mathcal{SEQ}^\tau$ ($\mathcal{S}eq^\tau$) , then there is a closed SPCF phrase M such that $\mathcal{SEQ}[\![\ \vdash M\,]\!]\ \bot = x$ ($\mathcal{S}eq[\![\ \vdash M\,]\!] \star \bot = x$).*

Proof. Due to its length, we have subdivided the proof into several parts.

Generalizing the Claim. The proof of the lemma is a lexicographic induction on the *depth* of the type τ and the cardinality of the element x, where the *depth* of ground type o is 1, the *depth* of $\tau_1 \to \ldots \tau_k \to o$ is $1 + \max\{depth(\tau_i) \,|\, 1 \leq i \leq k\}$. The term M that we shall construct will have the shape

$$\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.L$$

for some term L. Given the knowledge of how to construct L' for some x' below x, we will need to construct a term L that is like L' except for the code that accounts for the differences between x and x'. Put differently, we will need to keep track of places in the code that are due to occurrences of $\bot$ in some tree x' so that we know how to modify the code for extensions of x' that replace these occurrences of $\bot$. To accomplish this bookkeeping task, we will use a partitioning of M into contexts and holes filled with Ω. The generalized induction hypothesis is as follows:

> Given a finite tree x in the semantic domain associated with the type $\tau = \tau_1 \to \ldots \tau_k \to o$, and an arbitrary finite set Q of open queries, *i.e.*, $Q \subseteq Open(x)$, there exists a term
>
> $$M = \lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.L$$
>
> such that there is an injection from Q to the set of contexts with a single hole, satisfying the following condition:
>
> If $q \in Q$ is associated with $C[\ ]$, then
>
> $$M = C[\Omega^o]$$
>
> and for all terms N (of ground type) whose free variables are in $x_1^{\tau_1}, \ldots, x_k^{\tau_k}$, for all finite $x_i \in \mathcal{SEQ}^{\tau_i}$ (where $1 \leq i \leq k$),
>
> $$\mathcal{SEQ}[\![\ \vdash C[N]\,]\!]\ \bot \star x_1 \star \ldots \star x_k = \begin{cases} \mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k\rangle \\ \qquad \text{if } \|\pi_i(q)\| \sqsubseteq x_i, \text{ for all } i \\ x \star x_1 \star \ldots \star x_k \\ \qquad \text{if } \|\pi_i(q)\| \not\sqsubseteq x_i, \text{ for some } i \end{cases} \tag{‡}$$

Given an element *x* and its term *M*, we say that *M* can *grow in the direction of* all $q \in Q$.

The condition (‡) has a simple intuitive explanation: If *M* is applied to arguments that have responses to all the queries on the path from the root of x to the occurrence of $\perp$ represented by $C[\ ]$, then the only relevant part of the code is the filler of the hole; otherwise, the application has the same meaning as an application of x to the arguments. The condition implies that *M* represents *x*. To see this, set $N = \Omega$. Since $M = C[\Omega]$ the claim clearly holds if for some *i*, $\|\pi_i(q)\| \not\sqsubseteq x_i$. Thus, assume $\|\pi_i(q)\| \sqsubseteq x_i,$ for all i. Then

$$\begin{array}{rl} & \mathcal{SEQ}[\![\, \vdash M \,]\!] \perp \star x_1 \star \ldots \star x_k \\ = & \mathcal{SEQ}[\![\, \vdash C[\Omega] \,]\!] \perp \star x_1 \star \ldots \star x_k \\ = & \mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash \Omega \,]\!] \langle x_1, \ldots, x_k \rangle \\ = & \perp . \end{array}$$

On the other hand, it is also straightforward to see that

$$x \star x_1 \star \ldots \star x_k = \perp .$$

For the application to contain $? \cdot n$ or $? \cdot \mathsf{e}$, the element *x* would have to have a path r such that $\|\pi_i(r)\| \sqsubseteq x_i$ for all i and $(\pi_2^{\Rightarrow})^k(q) = ? \sqsubset (\pi_2^{\Rightarrow})^k(r)$, which by a *k*-fold application of Lemma 6.13 would imply $q \sqsubset r$ in contradiction to $q \in \mathit{Open}(x)$. By extensionality we may conclude that

$$\mathcal{SEQ}[\![\, M \,]\!] \perp = x.$$

Base Case. If x is empty, a valid representation is

$$M = \lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.\Omega.$$

The only open query, $q = \langle ? \rangle$, for *x* maps to $\lambda x_1^{\tau_1} \ldots x_k^{\tau_k}.[\ ]$. The injection obviously satisfies condition (‡).

Induction Step. If x is not empty and $Q \subseteq \mathit{Open}(x)$, we can pick an x' that is immediately below x, *i.e.*, there is some x' such that for some query $q' \in \mathit{Open}(x')$ and datum d,

$$x = x' \cup \{q' \cdot d\}.$$

Let $Q^- = \{q \in Q \,|\, q' \sqsubseteq q\}$ and let $Q^+ = Q \setminus Q^- \cup \{q'\}$. By inductive hypothesis, some term M' represents x' relative to the set Q^+, q' is associated with a unique context $C[\ ]$ that satisfies condition (‡) and, in particular, $M' = C[\Omega]$. Clearly, to turn into M, M' must grow in the direction of q'. But, to determine the precise replacement for the Ω term in the hole of $C[\ ]$, we need to give consideration to the two possible origins of the datum d.

Induction Step: Numbers and Errors. First, if $d = \langle n \rangle$ or $d = \mathsf{e}$, set

$$M = C[\ulcorner n \urcorner] \text{ or } M = C[\mathsf{e}],$$

respectively. To prove the appropriateness of this choice, we concentrate on the first case; the proof for the error case proceeds in an analogous fashion.

Since $q' \cdot \langle n \rangle$ is maximal in x, any $q \in Q$ is also in Q^+. By the inductive hypothesis for x' and Q^+, there is a context C' with two holes such that: (i) $\quad C[\ ] = C'[\ ]_1[\Omega]_2$ and therefore $M' = C'[\Omega]_1[\Omega]_2$; and (ii) q is associated with the context $C'[\Omega]_1[\ ]_2$. We claim that for x and Q, q is associated with $C'[\ulcorner n \urcorner]_1[\ ]_2$.

Associating $C'[\ulcorner n \urcorner]_1[\ ]_2$ with q satisfies (‡). We analyze each of the two parts of the constraint individually. Let N be an expression of ground type whose free variables are among $x_1^{\tau_1}, \ldots, x_k^{\tau_k}$. Let $x_1 \in \mathcal{SEQ}^{\tau_1}, \ldots, x_k \in \mathcal{SEQ}^{\tau_k}$ be the denotations of these variables.

1. Suppose that $\|\pi_i(q)\| \sqsubseteq x_i$ for all i. We need to prove that for all N,

$$\mathcal{SEQ}[\![\ \vdash C'[\ulcorner n \urcorner]_1[N]_2\,]\!] \perp \star x_1 \star \ldots \star x_k = \mathcal{SEQ}[\![\ \vdash x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle.$$

By inductive hypothesis for x' and Q^+, mapping q to $C'[\Omega]_1[\ ]_2$ satisfies (‡) and, thus,

$$\mathcal{SEQ}[\![\ \vdash C'[\Omega]_1[N]_2\,]\!] \perp \star x_1 \star \ldots \star x_k = \mathcal{SEQ}[\![\ \vdash x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle$$

for arbitrary N. By induction on the types τ_1 through τ_k, whose depth is smaller than τ''s depth, we can find terms $P_1, \ldots, P_k$ denoting $x_1, \ldots, x_k$. Hence,

$$\begin{array}{rl} & \mathcal{SEQ}[\![\ \vdash (C'[\Omega]_1[N]_2\ P_1 \ldots P_k)\,]\!] \perp . \\ = & \mathcal{SEQ}[\![\ \vdash C'[\Omega]_1[N]_2\,]\!] \perp \star x_1 \star \ldots \star x_k \\ = & \mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle \end{array}$$

Thus, for N, N' with free variables among $x_1^{\tau_1} \ldots x_k^{\tau_k}$, then, by monotonicity,

$$\mathcal{SEQ}[\![\ \vdash C'[N']_1[N]_2\,]\!] \perp \star x_1 \star \ldots \star x_k = \mathcal{SEQ}[\![\ \vdash x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle$$

if $\mathcal{SEQ}[\![\ \vdash x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle \in \mathbb{N}$.

If N denotes $\perp$ in the given environment, the equation also implies that

$$\mathcal{SEQ}[\![\ \vdash C'[\Omega]_1[\Omega]_2\,]\!] \perp \star x_1 \star \ldots \star x_k = \perp,$$

By the sequentiality of SPCF (see Theorem 7.5), the second hole of $C'[\ ]_1[\ ]_2$ is the syntactic sequentiality index of the program. Hence,

$$\mathcal{SEQ}[\![\ \vdash (C'[N']_1[N]_2\ P_1 \ldots P_k)\,]\!] \perp = \mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle = \perp$$

for any N' and, in particular, for $N' = \ulcorner n \urcorner$. In conclusion, for all possible N,

$$\mathcal{SEQ}[\![\ \vdash (C'[\ulcorner n \urcorner]_1[N]_2\ P_1 \ldots P_k)\,]\!] \perp = \mathcal{SEQ}[\![\, x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash N\,]\!]\langle x_1, \ldots, x_k \rangle,$$

which is the desired conclusion.

2. Suppose that $\|\pi_i(q)\| \not\sqsubseteq x_i$ for some i. In this case, the goal is to prove

$$\mathcal{SEQ}[\![\ \vdash C'[\ulcorner n \urcorner]_1[N]_2\,]\!] \perp \star x_1 \star \ldots \star x_k = x \star x_1 \star \ldots \star x_k.$$

Depending on whether or not the information in the arguments demands the evaluation of the contents of the first hole, which is associated with q', we must distinguish two subcases:

(a) If $\|\pi_i(q')\| \sqsubseteq x_i$ for all i, then by inductive hypothesis for x' and Q^+, the association of q' with $C'[\ ]_1[\Omega]$ satisfies condition $(\ddagger)$:

$$\begin{aligned}\mathcal{SEQ}[\![\ \vdash C'[\ulcorner n \urcorner]_1[\Omega]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k &= \mathcal{SEQ}[\![\ x_1^{\tau_1} \ldots x_k^{\tau_k} \vdash \ulcorner n \urcorner\ ]\!]\langle x_1, \ldots, x_k\rangle \\ &= \{? \cdot n\} \\ &= (x' \cup \{q' \cdot \langle n \rangle\}) \star x_1 \star \ldots \star x_k \\ &= x \star x_1 \star \ldots \star x_k.\end{aligned}$$

The proof goal now follows from monotonicity.

(b) Assume $\|\pi_i(q')\| \not\sqsubseteq x_i$ for some i. By the inductive hypothesis for x' and Q^+ the association of q' with $C'[\ ]_1[\Omega]$ as well as the association of q with $C'[\Omega][\ ]_2$ satisfy condition $(\ddagger)$. Thus,

$$\mathcal{SEQ}[\![\ C'[\Omega]_1[N]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k = x' \star x_1 \star \ldots \star x_k \tag{$*$}$$

$$\mathcal{SEQ}[\![\ \vdash C'[L]_1[\Omega]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k = x' \star x_1 \star \ldots \star x_k \tag{$**$}$$

for all N and L (whose free variables are among $x_1^{\tau_1}, \ldots, x_k^{\tau_k}$). Moreover, by a simple argument,

$$\begin{aligned}x' \star x_1 \star \ldots \star x_k &= (x' \cup \{q' \cdot \langle n \rangle\}) \star x_1 \star \ldots \star x_k \\ &= x \star x_1 \star \ldots \star x_k\end{aligned}$$

The rest depends on the result of $x \star x_1 \star \ldots \star x_k$:

i. $x \star x_1 \star \ldots \star x_k \neq \perp$: Equation $(*)$ implies the result by monotonicity.

ii. $x \star x_1 \star \ldots \star x_k = \perp$: By instantiating N and L in equations $(*)$ and $(**)$ appropriately, we can determine three equations about the program:

$$\begin{aligned}\mathcal{SEQ}[\![\ \vdash C'[\Omega]_1[\mathsf{e}]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k &= \perp \\ \mathcal{SEQ}[\![\ \vdash C'[\mathsf{e}]_1[\Omega]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k &= \perp \\ \mathcal{SEQ}[\![\ \vdash C'[\Omega]_1[\Omega]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k &= \perp\end{aligned}$$

By manifest sequentiality and the first two equations, neither hole in the context $C'[\ ]_1[\ ]_2$ can be a sequentiality index. By sequentiality and the third equation, for all L and N:

$$\mathcal{SEQ}[\![\ \vdash C'[L]_1[N]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k = \perp,$$

In particular, for all N,

$$\mathcal{SEQ}[\![\ \vdash C'[\ulcorner n \urcorner]_1[N]_2\ ]\!]\perp \star x_1 \star \ldots \star x_k = \perp = x \star x_1 \star \ldots \star x_k.$$

This concludes the proof that for any $q \in Q$, we have a context that satisfies $(\ddagger)$, which shows that the inductive claim holds for M, x, and the injection of queries from Q to contexts.

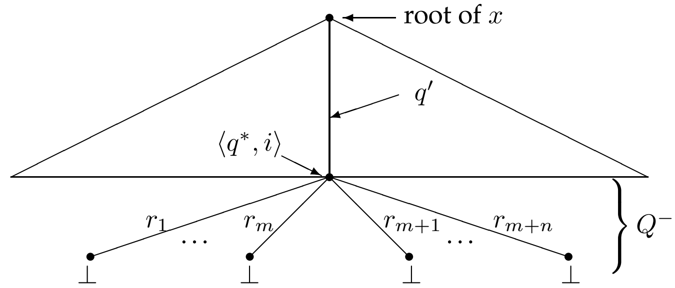


Figure 6: The Induction Step (Subtrees) of the Representability Lemma

Induction Step: Subtrees. Second, $d = \langle q^*, i\rangle$, *i.e.*, d is a query about the *i*th argument. Let $q_1, \ldots, q_{m+n}$ be all the queries in Q that dominate $q' \cdot \langle q^*, i\rangle$; they are the ones that contribute to the construction of *M*. By the construction of the exponent *sds*, all of these queries extend $q' \cdot \langle q^*, i\rangle$ with a response:

$$\begin{array}{rcl} q_1 & = & q' \cdot \langle q^*, i\rangle \cdot \langle r_1, i\rangle \\ \ldots & & \\ q_m & = & q' \cdot \langle q^*, i\rangle \cdot \langle r_m, i\rangle \\ q_{m+1} & = & q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+1}, i\rangle \\ \ldots & & \\ q_{m+n} & = & q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+n}, i\rangle \end{array}$$

The are two distinct classes of responses to $\langle q^*, i\rangle$: $q_1, \ldots, q_m$ and $q_{m+1}, \ldots, q_{m+n}$. The first *m* responses are *query responses*, that is, the *i*th argument demands more information about its possible inputs. Assuming that

$$\tau_i = \sigma_1 \to \ldots \sigma_l \to o \qquad \text{for } l \geq 0,$$

the responses have the shape

$$\begin{array}{rcl} r_1 & = & q^* \cdot \langle p_1, i_1\rangle \\ & \ldots & \\ r_m & = & q^* \cdot \langle p_m, i_m\rangle \end{array}$$

where $p_1, \ldots, p_m$ are queries for arguments $i_1, \ldots, i_m$, respectively, and $1 \leq i_1, \ldots, i_m \leq l$. The responses in the second set are *final responses*. They correspond to cases in which the *i*th argument does not demand more information about its arguments and simply returns an answer in $\mathbb{N}$. They have the form:

$$\begin{array}{rcl} r_{m+1} & = & q^* \cdot \langle a_1\rangle \\ & \ldots & \\ r_{m+n} & = & q^* \cdot \langle a_n\rangle \end{array}$$

where $a_1, \ldots, a_n \in \mathbb{N}$. The picture in Figure 6 provides a concise overview of the situation.

Now we need to construct a term N^* such that $C[N^*]$ denotes x. By the inductive hypothesis for x' (and Q^+), we know that if $x_1, \ldots, x_k$, the arguments to M, dominate the information in q', then the meaning of M is the meaning of N^* and the query q^* is open or answered in x_i. The program should then probe the ith argument and determine which of the $m+n$ responses x_i dominates, if any. Based on this probe, N^* can perform a dispatch according to the answer and diverge by evaluating Ω. Thus, a rough outline of N^* is

$$\begin{array}{lll}
\mathbf{let}\ w = \text{“probe argument } x_i\text{''}\ \mathbf{in} & & \\
\quad (\mathsf{if0}\ \text{“}w \text{ indicates } r_1 \in x_i\text{''}\ \Omega & ; & [1] \\
\quad \ldots & ; & \\
\quad (\mathsf{if0}\ \text{“}w \text{ indicates } r_m \in x_i\text{''}\ \Omega & ; & [m] \\
\quad (\mathsf{if0}\ \text{“}w \text{ indicates } r_{m+1} \in x_i\text{''}\ \Omega & ; & [m+1] \\
\quad \ldots & ; & \\
\quad (\mathsf{if0}\ \text{“}w \text{ indicates } r_{m+n} \in x_i\text{''}\ \Omega & ; & [m+n] \\
\quad \Omega)\ldots))\ldots) & ; &
\end{array}$$

where the expression ($\mathbf{let}\ w = L\ \mathbf{in}\ K$) abbreviates $((\lambda w.K)\ L)$. The first $m+n$ new Ω terms in $C[N^*]$ correspond to the $m+n$ new open queries in whose direction M can grow. Specifically, the Ω term on line $[j]$ indicates where M must grow for a tree y that extends x in the direction of the query $q' \cdot \langle q^*, i\rangle\langle r_j, i\rangle$ for $j, 1 \leq j \leq m+n$.

A naïve way of probing x_i is to apply it to the arguments $\|\pi_1(q^*)\|, \ldots, \|\pi_l(q^*)\|$, that is, the information in q^* about the arguments of x_i. Since σ_i, the type of $\|\pi_i(q^*)\|$, is smaller than τ, the inductive hypothesis implies that there is a term A_i that represents $\|\pi_i(q^*)\|$. Hence a syntactic representation of the probe expression would be the application

$$(x_i^{\tau_i}\ A_1\ \ldots\ A_l).$$

As long as x_i dominates a final response, say, r_{m+j} for $1 \leq j \leq n$, the naïve way of probing x_i works. The probe will evaluate to a_j (in $\mathbb{N}$), and it is easy to dispatch to the right branch in the following $\mathsf{if0}$-expression. However, if x_i contains some query response, say, $r_j = q^* \cdot \langle p_j, i_j\rangle$, then it ends up issuing a query asking for more information about its i_jth argument. Since the expression A_{i_j} only encodes the information embedded in q^*, *i.e.*, $p_j \in \mathit{Open}(\pi_{i_j}(q^*))$, this last action will cause the entire probe to diverge.

To get around the “over-exploration” of arguments to x_i, we need to define variants $B_1, \ldots, B_l$ of the arguments $A_1, \ldots, A_l$ such that each variant B_h initiates a non-local exit from the application $(x_i^{\tau_i}\ B_1\ \ldots\ B_l)$ whenever x_i asks for information in the direction of any of the responses $r_1, \ldots, r_m$. In SPCF the catch operator, when applied to an appropriate procedure, can simulate a non-local exit. Thus, to implement our ideas in SPCF, the expression $(x_i^{\tau_i}\ B_1\ \ldots\ B_l)$ needs to be parameterized such that it demands its jth argument precisely when x_i dominates r_j. Call the resulting procedure $N^@$. Then by applying the catch operator to $N^@$, the program can determine which query or which final response x_i dominates and can thus branch to the correct line in N^*.

Expressed formally, we must define a λ-abstraction

$$N^@ = \lambda y_1^o \ldots y_m^o.(x_i^{\tau_i}\ B_1 \ldots B_l)$$

that satisfies the following condition:

$$\mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash N^{@} \,]\!]\langle x_1, \ldots, x_k\rangle = \begin{cases} \{\langle ?\rangle \cdot \langle ?, j\rangle\} & \text{if } r_j \in x_i \\ & \text{for some } j, 1 \le j \le m \\ \{\langle ?\rangle \cdot \langle a\rangle\} & \text{if } q^* \cdot \langle a\rangle \in x_i, a \in \mathbb{N} \end{cases}$$

Then:

$$\mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@}) \,]\!]\langle x_1, \ldots, x_k\rangle = \begin{cases} \{? \cdot (j-1)\} & \text{if } r_j \in x_i \\ \{? \cdot (a+m)\} & \text{if } q^* \cdot \langle a\rangle \in x_i \\ \{? \cdot \mathsf{e}\} & \text{if } q^* \cdot \mathsf{e} \in x_i \\ \bot & \text{otherwise} \end{cases}$$

The last two parts are implied by sequentiality and manifest sequentiality.

The key to constructing $N^{@}$ is defining the expressions $B_1, \ldots, B_l$. Each expression B_h must evaluate the variable y_j^o if $r_j \in x_i$. In addition, each expression B_h must satisfy the constraint $\|\pi_h(q^*)\| \sqsubseteq \mathcal{SEQ}[\![\, y_1^o, \ldots, y_s^o \vdash B_h \,]\!]\langle y_1, \ldots, y_s\rangle$ for arbitrary $y_1, \ldots, y_s$. Since the construction of the expression is complicated, we assume for the moment that we can construct the argument expressions $B_1, \ldots, B_l$ and continue with the rest of the construction of $N^{@}$.

Given the expression $N^{@}$, N^* is easy to construct. Since the application $(\mathsf{catch}\ N^{@})$ yields an encoding of the index j of x_i's response r_j (if any), N^* can branch to the correct line representing the query q_j by performing a sequential case split. Without loss of generality, we assume that the final answers are sorted: $a_1 < a_2 < \ldots < a_n$. Then

$$\begin{array}{rcll} N^* & = & \textbf{let } w = (\mathsf{catch}\ N^{@}) \textbf{ in} & \\ & & \quad (\mathsf{if0}\ w\ \Omega & ;\ [1] \\ & & \quad \ldots & \\ & & \quad (\mathsf{if0}\ (\mathsf{sub1}^{m-1}\ w)\ \Omega & ;\ [m] \\ & & \quad (\mathsf{if0}\ (\mathsf{sub1}^{a_1+m}\ w)\ \Omega & ;\ [m+1] \\ & & \quad \ldots & \\ & & \quad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m}\ w)\ \Omega & ;\ [m+n] \\ & & \quad \Omega)\ldots))\ldots) & \end{array}$$

where the expression $(\mathsf{sub1}^b\ L)$ abbreviates the b-fold application of $\mathsf{sub1}$ to the argument expression L.

Beyond the construction of M the inductive hypothesis also demands the definition of an injection from Q to the set of contexts derived from M such that the injection satisfies (‡). If $q \in Q^+$ then we can exploit the inductive hypothesis for x' and Q^+ and use the injection from Q^+ to contexts for M' to find an appropriate context for q and M; the proof that this association works is only a minor variant of the proof in the first subcase of the induction step. Otherwise, by assumption, q is one of $q_1, \ldots, q_{m+n}$. Say, $q = q_{m+j}$ for $1 \le j \le n$ and $r_{m+j} = q^* \cdot \langle a_j\rangle$. If C_{m+j} is the context whose hole contains the Ω term on line $m+j$ of N^*, *i.e.*, $C[N^*] = C[C_{m+j}[\Omega]]$, then $C[C_{m+j}[\ ]]$ is the context associated with q.

We will concentrate on a query that ends in a final response to simplify the presentation: q_{m+j} for $1 \le j \le n$ and $r_{m+j} = q^* \cdot \langle a_j\rangle$. For a response that ends in a query (r_j for $1 \le j \le m$), the proof proceeds in essentially the same manner.

Associating $C[C_{m+j}[\ ]]$ with q satisfies (‡). We analyze each of the two parts of condition (‡) separately. Let N be a term with free variables in $x_1^{\tau_1}, \ldots, x_k^{\tau_k}$, and let $x_1 \in \mathcal{SEQ}^{\tau_1}$, $\ldots$, $x_k \in \mathcal{SEQ}^{\tau_k}$ be the denotations of these variables.

1. Suppose that $\|\pi_h(q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+j}, i\rangle)\| \sqsubseteq x_h$ for all h, which clearly means $\|\pi_h(q')\| \sqsubseteq x_h$ and at index i implies

$$\|\pi_i(q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+j}, i\rangle)\| = \|\pi_i(q')\| \cup \{r_{m+j}\} \sqsubseteq x_i.$$

   We must show that the meaning of $C[C_{m+j}[N]]$ is the meaning of N:

$$\begin{array}{ll} & \mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C[C_{m+j}[N]] \,]\!]\langle x_1, \ldots, x_k\rangle \\ = & \mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash N \,]\!]\langle x_1, \ldots, x_k\rangle \end{array}$$

   By the inductive hypothesis for x' and Q^+, q' is associated with $C[\ ]$ and satisfies (‡). Since $\|\pi_h(q')\| \sqsubseteq x_h$ for all h by assumption,

$$\begin{array}{ll} & \mathcal{SEQ}[\![\, \vdash C[C_{m+j}[N]] \,]\!]\ \bot \star x_1 \star \ldots \star x_k \\ = & \mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N] \,]\!]\langle x_1, \ldots, x_k\rangle. \end{array}$$

   Also by assumption, $r_{m+j} = q^* \cdot \langle a_j\rangle$ for $a_j \in \mathbb{N}$ and thus,

$$\mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@}) \,]\!]\langle x_1, \ldots, x_k\rangle = \{? \cdot (a_j + m)\}.$$

   The rest is a straightforward calculation:

$$\begin{array}{lll} & \mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N] \,]\!]\langle x_1, \ldots, x_k\rangle & \\ = & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash & \\ & \qquad (\mathsf{if0}\ w\ \Omega \ldots & \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{m-1}\ w)\ \Omega & \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_1+m}\ w)\ \Omega & \\ & \qquad \ldots & \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_j+m}\ w)\ N & \\ & \qquad \ldots & \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m}\ w)\ \Omega)\ldots)\ldots))\ldots)]\!]\langle x_1, \ldots, x_k, \{? \cdot (a_j + m)\}\rangle & \\ = & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash & \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_j+m}\ w)\ N & \\ & \qquad \ldots & \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m}\ w)\ \Omega)\ldots)]\!]\langle x_1, \ldots, x_k, \{? \cdot (a_j + m)\}\rangle & \\ = & \mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash N \,]\!]\langle x_1, \ldots, x_k, \{? \cdot (a_j + m)\}\rangle & \\ = & \mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash N \,]\!]\langle x_1, \ldots, x_k\rangle & \text{because } w^o \text{ is not free in } N. \end{array}$$

2. Suppose that $\|\pi_l(q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+j}, i\rangle)\| \not\sqsubseteq x_l$ for some l. Now we must show that the meaning of $C[C_{m+j}[N]]$ is independent of N:

$$\mathcal{SEQ}[\![\, \vdash C[C_{m+j}[N]] \,]\!]\ \bot \star x_1 \star \ldots \star x_k = x \star x_1 \star \ldots \star x_k.$$

   We distinguish two subcases:

(a) If for any h (unrelated to l) $\|\pi_h(q')\| \not\sqsubseteq x_h$ then by the inductive hypothesis for x' and Q^+, for any K,

$$\mathcal{SEQ}[\![\vdash C[K]]\!] \perp \star x_1 \star \ldots x_k = x' \star x_1 \star \ldots \star x_k.$$

By instantiating with $C_{m+j}[L]$,

$$\begin{array}{rl} & \mathcal{SEQ}[\![\vdash C[C_{m+j}[N]]]\!] \perp \star x_1 \star \ldots \star x_k \\ = & x' \star x_1 \star \ldots \star x_k \\ = & (x' \cup \{q' \cdot \langle q^*, i\rangle\}) \star x_1 \star \ldots \star x_k \\ = & x \star x_1 \star \ldots \star x_k \end{array}$$

The last two steps hold because $\|\pi_h(q')\| \not\sqsubseteq x_h$ and therefore $q' \cdot \langle q^*, i\rangle$ does not contribute anything to the final output of the application.

(b) If $\|\pi_h(q')\| \sqsubseteq x_h$ for all h then obviously, $\|\pi_i(q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+j}, i\rangle)\| \not\sqsubseteq x_i$. Moreover, the inductive hypothesis for x' and Q^+ implies that for any term in the hole of $C[\ ]$, and in particular for $C_{m+j}[L]$,

$$\begin{array}{rl} & \mathcal{SEQ}[\![\vdash C[C_{m+j}[N]]]\!] \perp \star x_1 \star \ldots \star x_k \\ = & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]]\!]\langle x_1, \ldots, x_k\rangle. \end{array}$$

The rest of this case depends on the response of x_i to the query q^*, if any:

i. If $q^* \in \mathit{Open}(x_i)$ then

$$\begin{array}{rl} & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]]\!]\langle x_1, \ldots, x_k\rangle \\ = & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash (\mathsf{if0}\ w\ \Omega\ \ldots)]\!]\langle x_1, \ldots, x_k, \perp\rangle \\ = & \perp \end{array}$$

because $\mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@})]\!]\langle x_1, \ldots, x_k, \perp\rangle = \perp$.

ii. If $q^* \cdot \mathsf{e} \in x_i$ (for $\mathsf{e} \in \mathbb{E}$), then

$$\begin{array}{rl} & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]]\!]\langle x_1, \ldots, x_k\rangle \\ = & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash (\mathsf{if0}\ w\ \Omega\ \ldots)]\!]\langle x_1, \ldots, x_k, \{? \cdot \mathsf{e}\}\rangle \\ = & \{? \cdot \mathsf{e}\} \end{array}$$

because $\mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@})]\!]\langle x_1, \ldots, x_k\rangle = \{? \cdot \mathsf{e}\}$ if $q^* \cdot \mathsf{e} \in x_i$.

iii. If $q^* \cdot \langle a\rangle \in x_i$, then two cases are possible. First, $a = a_h$ for some $h, 1 \leq h \leq n$ but $h \neq j$. Then because $\mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@})]\!]\langle x_1, \ldots, x_k\rangle = \{? \cdot (a_j + m)\}$,

$$\begin{array}{rl} & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]]\!]\langle x_1, \ldots, x_k\rangle \\ = & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash \end{array}$$

$$\begin{array}{rl}
 & \qquad (\mathsf{if0}\ w\ \Omega \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{m-1}\ w)\ \Omega \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_1+m}\ w)\ \Omega \\
 & \qquad \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_j+m}\ w)\ N \\
 & \qquad \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m}\ w)\ \Omega) \ldots) \ldots)) \ldots)\rrbracket \\
 & \quad \langle x_1, \ldots, x_k, \{? \cdot (a_h + m)\}\rangle \\
= & \mathcal{SEQ}\llbracket x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_h+m}\ w)\ \Omega \\
 & \qquad \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m}\ w)\ \Omega) \ldots)\rrbracket \langle x_1, \ldots, x_k, \{? \cdot (a_j + m)\}\rangle \\
= & \mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash \Omega\,\rrbracket \langle x_1, \ldots, x_k, \{? \cdot (a_h + m)\}\rangle \\
= & \perp .
\end{array}$$

Second, a may be distinct from $a_1, \ldots, a_n$ such that, without loss of generality, $a < a_n$. Again, one of the test expressions will diverge:

$$\begin{array}{rl}
 & \mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]\,\rrbracket \langle x_1, \ldots, x_k\rangle \\
= & \mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash (\mathsf{if0}\ w\ \Omega\ \ \ldots)\,\rrbracket \langle x_1, \ldots, x_k, \{? \cdot a\}\rangle \\
= & \mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash (\mathsf{if0}\ (\mathsf{sub1}^{a_j+m}\ w)\ \Omega\ \ \ldots)\,\rrbracket \langle x_1, \ldots, x_k, \{? \cdot a\}\rangle \\
 & \text{for } a_j > a \\
= & \perp
\end{array}$$

If $a > a_n$, then the evaluation will reach the else branch of the last if0-expression, which will force divergence again.

iv. If $q^* \cdot \langle p^*, 1\rangle \in x_i$ for some query p^*, we must also consider two cases. First, p^* may be one of $p_1, \ldots, p_m$, *e.g.*, $p^* = p_h$ for $h, 1 \leq h \leq m$. This implies that

$$\mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@})\,\rrbracket = \{? \cdot (h-1)\}.$$

A simple calculation now shows that

$$\begin{array}{rl}
 & \mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]\,\rrbracket \langle x_1, \ldots, x_k\rangle \\
= & \mathcal{SEQ}\llbracket x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash \\
 & \qquad (\mathsf{if0}\ w\ \Omega \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{h-1}\ w)\ \Omega \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_1+m}\ w)\ \Omega \\
 & \qquad \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_j+m}\ w)\ N \\
 & \qquad \ldots \\
 & \qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m}\ w)\ \Omega) \ldots) \ldots)) \ldots)\rrbracket \\
 & \quad \langle x_1, \ldots, x_k, \{? \cdot (h-1)\}\rangle \\
= & \mathcal{SEQ}\llbracket\, x_1^{\tau_1}, \ldots, x_k^{\tau_k}, w^o \vdash (\mathsf{if0}\ (\mathsf{sub1}^{h-1}\ w)\ \Omega \ldots)\,\rrbracket
\end{array}$$

$$\langle x_1, \ldots, x_k, \{? \cdot (h-1)\}\rangle$$
$$= \quad \perp$$

Second, p^* may be distinct from any of the $p_1, \ldots, p_m$, in which case

$$\mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (\mathsf{catch}\ N^{@})\,]\!]\langle x_1, \ldots, x_k\rangle = \perp,$$

and therefore

$$\mathcal{SEQ}[\![\, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash C_{m+j}[N]\,]\!]\langle x_1, \ldots, x_k\rangle = \perp .$$

For all four cases, $C_{m+j}[N]$ in the environment $\langle x_1, \ldots, x_k\rangle$ evalutes as required:

$$\begin{array}{rcl} x \star x_1 \star \ldots \star x_k & = & (x' \cup \{q' \cdot \langle q^*, i\rangle\}) \star x_1 \star \ldots \star x_k \\ & = & \left\{ \begin{array}{ll} \perp & \text{if } q^* \in \mathit{Open}(x_i) \\ \{? \cdot \mathsf{e}\} & \text{if } q^* \cdot \mathsf{e} \in x_i \\ \perp & \text{if } q^* \cdot \langle a\rangle \in x_i, a \neq a_j \\ \perp & \text{if } q^* \cdot \langle p^*, 1\rangle \in x_i \end{array} \right. \end{array}$$

Given that N was arbitrary we were able to show that the association of $C_{m+j}[\ ]$ with $q = q' \cdot \langle q^*, i\rangle \cdot \langle r_{m+j}, i\rangle$ satisfies (‡).

Constructing $B_1, \ldots, B_l$. To complete the proof of the lemma, we must

- construct the argument expressions $B_1, \ldots, B_l$, and
- prove that the expression $N^{@}$, constructed from $B_1, \ldots, B_l$, satisfies the correctness conditions identified above.

Let h be an index in the range $1, \ldots, l$. The type of B_h has the form

$$\sigma_h = \nu_1 \to \ldots \nu_t \to o$$

for some integer $t \geq 0$. Its construction falls into one of two cases. In the first case, no query response r_j ($1 \leq j \leq m$) asks for more information about the hth argument. In this case, we simply define B_h as the closed expression representing the tree $\|\pi_h(q^*)\|$, which exists by the induction hypothesis because the type σ_h of B_h is smaller than τ.

In the other case, a finite number s of query responses r_j ask for more information about the hth argument. We can assume without loss of generality that these query responses are $r_1 = q^* \cdot \langle p_1, h\rangle$, ..., $r_s = q^* \cdot \langle p_s, h\rangle$, and that the variables $y_1^o, \ldots, y_s^o$ correspond to these responses. From the assumptions, $p_1, \ldots, p_s \in \mathit{Open}(\|\pi_h(q^*)\|)$. Now our goal is to construct a procedure B_h that requests information about y_j^o if x_i inquires about p_j ($1 \leq j \leq s$). Thus, B_h must be a procedure of type σ_h but its body must also depend on the parameters $y_1^o, \ldots, y_s^o$. If E_h is the body of B_h, the latter has the following shape:

$$B_h = \lambda z_1^{\nu_1} \ldots z_t^{\nu_t}.(E_h\ z_1^{\nu_1} \ldots z_t^{\nu_t}\ y_1^o\ \ldots\ y_s^o).$$

In other words, E_h is procedure of type

$$\sigma'_h = \nu_1 \rightarrow \ldots \nu_t \rightarrow \underbrace{o \rightarrow \ldots \rightarrow o}_{s-\text{times}} \rightarrow o$$

that acts like $\|\pi_h(q^*)\|$ with respect to the first t arguments and contains a query for y^o_j if the first t arguments dominate the information in the path p_j $(1 \leq j \leq s)$.

Semantically speaking, we need to (1) inject $\|\pi_h(q^*)\|$ from the domain for σ_h into σ'_h and (2) extend $\|\pi_h(q^*)\|$ at the positions $p_1, \ldots, p_s$ so that it issues a query for argument $t+1, \ldots, t+s$, respectively:

1. Every z tree of type σ_h directly corresponds to a tree in σ'_h. By changing every initial address on a path in z from

   $$\langle \ldots \langle ?, \underbrace{2\rangle, \ldots, 2\rangle}_{t} \text{ to } \langle \ldots \langle ?, \underbrace{2\rangle, \ldots, 2\rangle}_{s+t}$$

   and every final datum from

   $$\langle \ldots \langle ? \cdot n, \underbrace{2\rangle, \ldots, 2\rangle}_{t} \text{ to } \langle \ldots \langle ? \cdot n, \underbrace{2\rangle, \ldots, 2\rangle}_{s+t},$$

   we get a tree in the extended domain. We use the notation $z^{\sigma'_h}$ for the result of injecting z. Since the tree in the extended domain does not contain any queries about the arguments $t+1, \ldots, t+s$, it is immediate that $\|\pi_h(q^*)\|^{\sigma_h}$ and $\|\pi_h(q^*)\|^{\sigma'_h}$ relate to each other as follows:

   $$\|\pi_h(q^*)\|^{\sigma_h} \star z_1 \star \ldots \star z_t = \|\pi_h(q^*)\|^{\sigma'_h} \star z_1 \star \ldots \star z_t \star y_1 \star \ldots \star y_s$$

   for all $z_1, \ldots, z_t$ and $y_1, \ldots, y_s$.

2. Given the tree $\|\pi_h(q^*)\|^{\sigma'_h}$, we can form a tree that grafts appropriate subtrees at (the paths in the extended domain) $p_1, \ldots, p_s$ onto the tree:

   $$e_h = \|\pi_h(q^*)\|^{\sigma'_h} \cup \{p_1^{\sigma'_h} \cdot \langle ?, t+1\rangle, \ldots, p_s^{\sigma'_h} \cdot \langle ?, t+s\rangle\}.$$

Interpreted as a function, e_h obviously has the desired behavior. The inductive hypothesis implies that there is a representing term E_h for e_h because σ'_h is smaller than τ.

The meaning of B_h depends on the values of $y^o_1, \ldots, y^o_s$ in the environment. Let $\vec{y} = \langle y_1, \ldots, y_s\rangle$ be a vector of s arbitrary values in $\mathbb{N}^e_\perp$. Then we claim that B_h denotes the tree

$$(\|\pi_h(q^*)\| \cup \{p_j \cdot y_j \mid \text{ if } y_j \in \mathbb{E}\})^{\sigma_h}$$

That is, if none of the variables $y^o_1, \ldots, y^o_s$ denotes an error value, B_h denotes $\|\pi_h(q^*)\|$. If some variable y^o_j stands for an error value y_j, the error value is propagated and is grafted onto the end of the query $p_j \in \mathit{Open}(\|\pi_h(q^*)\|)$. We refer to the meaning of B_h relative to $\vec{y}$ as $b^{\vec{y}}_h$.

To prove that the denotation of B_h is the above tree, assume that $z_1 \in \mathcal{SEQ}^{\nu_1}, \ldots, z_t \in \mathcal{SEQ}^{\nu_t}$ and $y_1, \ldots, y_s \in \mathbb{N}^e_\perp$ are arbitrary:

$$\begin{aligned}
& \mathcal{SEQ}[\![y_1^o \ \ldots \ y_s^o \vdash B_h]\!]\langle y_1, \ldots, y_s\rangle \star z_1 \star \ldots \star z_t \\
= \ & \mathcal{SEQ}[\![y_1^o \ \ldots \ y_s^o z_1^{\nu_1} \ldots z_t^{\nu_t} \vdash (E_h \ z_1^{\nu_1} \ldots z_t^{\nu_t} \ y_1^o \ \ldots \ y_s^o)]\!]\langle y_1, \ldots, y_s, z_1, \ldots, z_t\rangle \\
= \ & e_h \star z_1 \star \ldots \star z_t \star y_1 \star \ldots \star y_s \\
= \ & (\|\pi_h(q^*)\| \cup \{p_1 \cdot \langle ?, t+1\rangle, \ldots, p_s \cdot \langle ?, t+s\rangle\})^{\sigma'_h} \star z_1 \star \ldots \star z_t \star y_1 \star \ldots \star y_s \\
= \ & \begin{cases} \{? \cdot y_1\} & \text{if } \|\pi_g(p_1)\| \sqsubseteq z_g \text{ for } 1 \leq g \leq t, y_1 \in \mathbb{E} \\ \ldots & \\ \{? \cdot y_s\} & \text{if } \|\pi_g(p_s)\| \sqsubseteq z_g \text{ for } 1 \leq g \leq t, y_s \in \mathbb{E} \\ \|\pi_h(q^*)\| \star z_1 \star \ldots \star z_t \star y_1 \star \ldots \star y_s & \text{otherwise} \end{cases} \\
= \ & (\|\pi_h(q^*)\| \cup \{p_j \cdot y_j \mid \text{ if } y_j \in \mathbb{E}\})^{\sigma_h} \star z_1 \star \ldots \star z_t,
\end{aligned}$$

which, by extensionality, implies $\mathcal{SEQ}[\![y_1^o \ \ldots \ y_s^o \vdash B_h]\!]\langle y_1, \ldots, y_s\rangle = b_h^{\vec{y}}$.

After constructing and analyzing B_h, we must show that the denotation of

$$N^@ = \lambda y_1^o \ldots y_m^o.(x_i^{\tau_i} \ B_1 \ldots B_l)$$

is such that

$$\mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash N^@]\!]\langle x_1, \ldots, x_k\rangle = \begin{cases} \{\langle ?\rangle \cdot \langle a\rangle\} & \text{if } q^* \cdot \langle a\rangle \in x_i, a \in \mathbb{N} \\ \{\langle ?\rangle \cdot \langle ?, j\rangle\} & \text{if } r_j \in x_i \\ & \text{for some } j, 1 \leq j \leq m. \end{cases}$$

To prove this claim, assume that x_i contains a response to q^*, say, r_j. Let $y_1, \ldots, y_m$ be l arbitrary values from $\mathbb{N}^e_\perp$ and let $\vec{y}_1, \ldots, \vec{y}_l$ be vectors containing the values from $y_1, \ldots, y_m$ of all the free variables among $y_1^o, \ldots, y_m^o$ for the expressions $B_1, \ldots, B_l$, respectively. Then:

$$\begin{aligned}
& \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash N^@]\!]\langle x_1, \ldots, x_k\rangle \star y_1 \star \ldots \star y_m \\
= \ & \mathcal{SEQ}[\![x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash (x_i \ B_1 \ \ldots \ B_l)]\!]\langle x_1, \ldots, x_k, y_1, \ldots, y_m\rangle \\
= \ & x_i \star b_1^{\vec{y}_1} \star \ldots \star b_l^{\vec{y}_l} \\
= \ & \{(\pi_2^{\Rightarrow})^l(r_j) \mid r_j = q^* \cdot \langle a\rangle, a \in \mathbb{N}\} \cup \{(\pi_2^{\Rightarrow})^l(q_j) \cdot y_{h_j} \mid r_j = q^* \cdot \langle p_j, h_j\rangle, \text{ if } y_{h_j} \in \mathbb{E}\} \\
& \text{because } b_1^{\vec{y}} \sqsupseteq \|\pi_1(q^*)\|, \ldots, b_l^{\vec{y}} \sqsupseteq \|\pi_l(q^*)\| \\
= \ & \{? \cdot a \mid r_j = q^* \cdot \langle a\rangle, a \in \mathbb{N}\} \cup \{? \cdot y_{h_j} \mid r_j = q^* \cdot \langle p_j, h_j\rangle, y_{h_j} \in \mathbb{E}\} \\
& \text{since } p_j \cdot y_j \in b_{h_j}^{\vec{y}_{h_j}} \text{ if } y_{h_j} \in \mathbb{E} \text{ and } p_j \in \mathit{Open}(b_{h_j}^{\vec{y}_{h_j}}) \text{ if } y_{h_j} \in \mathbb{N}_\perp \\
= \ & \begin{cases} \{\langle ?\rangle \cdot \langle a\rangle\} \star y_1 \star \ldots \star y_m & \text{if } r_j = q^* \cdot \langle a\rangle \\ \{\langle ?\rangle \cdot \langle ?, j\rangle\} \star y_1 \star \ldots \star y_m & \text{if } r_j = q^* \cdot \langle p_j, h_j\rangle \end{cases}
\end{aligned}$$

By extensionality, the denotation of $N^@$ satisfies the desired condition. This observation completes the proof of the lemma for the semantics based on $\mathcal{SEQ}$.

With minor adjustments, the proof is valid for a semantics based on $\mathcal{S}eq$ as long as the set of error values is not empty. As for the non-extensional semantics $\mathcal{S}eq^\emptyset$, the proof proceeds as above. Since an error-free *x* is represented by a term *M* in SPCF(∅), and since by Lemma 7.3 terms in SPCF(∅) of type τ denote the same tree in $\mathcal{S}eq^\tau$ independently of the underlying error set, it is also true that $\mathcal{S}eq^\emptyset[\![M]\!]\star \perp = x$, *i.e.*, *M* denotes *x* in the intensional semantics. ∎

8 An Adequate Operational Semantics for SPCF

Although the denotational semantics of a programming language is an elegant tool for reasoning about the phrases of the language, an operational semantics offers a better basis for the derivation of an implementation. Moreover, when defined via the reductions of a calculus, an operational semantics also provides a simple equational theory that is useful for proving many observational equivalences.

To equip SPCF with an operational semantics, we follow Plotkin's program [25]—as extended by Felleisen, Friedman and Hieb [13, 14, 15]—on relating λ-calculi and abstract machines. More specifically, we define an extension of the typed λ-calculus and, based on it, a deterministic text rewriting machine for SPCF programs. We then prove an adequacy theorem for SPCF, *i.e.*, we show that for any program P, the meaning of P is n or e if and only if P reduces to $\ulcorner n \urcorner$ or e, respectively. Based on this adequacy result, the full abstraction theorem can now be formulated as follows: given two SPCF phrases M and N, $\mathcal{S}eq[\![M]\!] \sqsubseteq \mathcal{S}eq[\![N]\!]$ if and only if for for any program context $C[\ ]$ for M and N, $C[M]$ evaluates to w implies $C[N]$ evaluates to w where w is either a numeral or an error element.[17]

8.1 Operational Semantics

The description of the operational semantics relies on the notion of an *evaluation* context [13] for two separate purposes. First, the notion of an evaluation context is useful for defining notions of reduction that capture the behavior of error elements and control facilities like catch in a context-free manner. Second, given a reduction system, a *standard* reduction of a program is the repeated reduction of the redex that occurs in the hole of the evaluation context. This definition yields a deterministic textual rewriting machine because every SPCF program (that is not a value) has a unique partitioning into an evaluation context and a (proper) redex.

Definition 8.1 (*Evaluation Contexts*) The set of evaluation contexts is the following subset of the set of contexts:

$$E ::= [\ ] \mid (f\ E) \mid (E\ M) \mid \mathsf{catch}\ (\lambda x_1 \ldots x_n.E) \qquad \text{for } n \geq 0$$

If $E[\ ]$ is an evaluation context, then $(E[\ ])[N/x]$ denotes the context that results from capture-free substitution of all free *x* by N in the subterms of $E[\ ]$. ∎

The set of evaluation contexts is a subset of the set of contexts (see Definition 3.3). Generally speaking, the hole of an evaluation context can only occur where a reduction *must* take place due to the strictness properties of the primitive procedures and procedure application in SPCF. There are four specific situations that meet these criteria (in SPCF). First, a hole can clearly occur to the right of a primitive procedure because all primitives are strict. Second, in an application the only position that must be evaluated is the function position: hence, an evaluation context may have a hole in the function position of an arbitrary application. Third, since catch must determine whether or not its argument is constant or strict

[17] The results presented in this section account for the fact that $(\mathsf{sub1}\ \ulcorner 0 \urcorner)$ denotes bottom but it does not rely on it. It is easy to adapt the proofs for a version of SPCF in which $(\mathsf{sub1}\ \ulcorner 0 \urcorner)$ returns $\ulcorner 0 \urcorner$ or signals an error.

in some position, it must clearly evaluate its argument *and*, if the argument reduces to a procedure, the *body* of the argument's value.[18] To reflect the second part of this idea, an evaluation context may also have a hole in the procedure body of the argument to catch. Finally, to provide a base case for the inductive definition, we also include the plain hole in the set of evaluation contexts.

For the specification and analysis of the operational semantics we need to extend the substitution and context-filling operation to evaluation contexts. It is crucial that these operations, *i.e.*, filling an evaluation context with an evaluation context or replacing a free variable by an arbitrary term in an evaluation context, yield not only arbitrary contexts but evaluation contexts.

Lemma 8.2 *If E and E' are evaluation contexts, then so are $E[E'[\ ]]$ and $(E[\ ])[N/x]$.*

Proof. The proofs are simple inductions on the structure of E. ∎

Given the definition of evaluation contexts, we can now turn to the definition of the SPCF calculus. The calculus is an extension of the PCF calculus. First, the basic set of axioms contains the usual laws for procedure application (β) and for primitives (Y, $\mathsf{add1}, \mathsf{sub1}$, $\mathsf{if0}/0$, $\mathsf{if0}/1$). Second, the laws for catch and error elements reflect the denotational semantics. Accordingly a term signals an error if the error occurs in a position that must be evaluated. Put formally, an error element erases any surrounding evaluation context. Similarly, when a bound variable occurs in an evaluation context (in the procedure body), a catch application will signal that the procedure is strict in this argument. An application of catch to a procedure whose body is a numeral can be replaced by an appropriate result. Finally, there is no reduction for Ω^o because it represents the provably diverging program.

[18] Operationally speaking, **catch** acts like a (strict) **let**-expression for which the evaluator creates the values for the variables on the fly. The values are tags that mark the evaluation context of the **let**-expression. When one of the tags is touched during the evaluation of the body, the portion of the evaluation context between the markers and the occurrence of the tag needs to be erased. To avoid the introduction of an additional syntactic object, we employ variables as tags and use the notation **catch** $\lambda x_1 \ldots x_m.$ to mark the evaluation context at the appropriate place. In an abstract machine with an explicit control stack, like the CEK [13] machine, this implementation technique corresponds to marking the control stack (K) and binding the variables of the **catch** argument to these tags. When an environment lookup produces a tag, the control stack is erased to an appropriate extent.

Definition 8.3 (*SPCF Reductions, Calculus***)** The nine basic notions of reduction for SPCF are the following nine relations between SPCF phrases:

$$
\begin{array}{rcll}
\mathsf{add1}\ \ulcorner n \urcorner & \longrightarrow & \ulcorner n+1 \urcorner & (\mathsf{add1}) \\
\mathsf{sub1}\ \ulcorner n+1 \urcorner & \longrightarrow & \ulcorner n \urcorner & (\mathsf{sub1}) \\
\mathsf{if0}\ \ulcorner 0 \urcorner & \longrightarrow & \lambda xy.x & (\mathsf{if0/0}) \\
\mathsf{if0}\ \ulcorner n+1 \urcorner & \longrightarrow & \lambda xy.y & (\mathsf{if0/1}) \\
\mathsf{Y} M & \longrightarrow & M(\mathsf{Y}\ M) & (\mathsf{fix}) \\
(\lambda x.M)\ N & \longrightarrow & M[N/x] & (\beta) \\
E[\mathsf{e}] & \longrightarrow & \mathsf{e}, \qquad E \neq [\ ], \mathsf{e} \in \mathbb{E} & (\mathsf{error}) \\
\mathsf{catch}\ (\lambda x_1 \ldots x_m.\ulcorner n \urcorner) & \longrightarrow & \ulcorner m+n \urcorner & (\mathsf{return}) \\
\mathsf{catch}\ (\lambda x_1 \ldots x_m.E[x_i]) & \longrightarrow & \ulcorner i-1 \urcorner \qquad \text{where } i \leq m & (\mathsf{catch})
\end{array}
$$

Additional Constraints: First, in (error), $E[\mathsf{e}]$ must be of ground type. Second, the last rule assumes that given a term of the shape $E[x_i]$, the occurrence of x_i in the hole of E is a free occurrence.
Terminology: As always, we refer to the left hand side of one of the above rules as "redex". $E[\mathsf{e}]$ is called an *error redex*; other redexes are proper redexes.

The SPCF calculus is the equational theory generated from the nine axioms above using the inference rules:

$$
\frac{M \longrightarrow N}{M = N} \qquad \frac{M = N}{C[M] = C[N]} \qquad \frac{M = N \quad N = L}{M = L}
$$

$$
\frac{}{M = M} \qquad \frac{M = N}{N = M}
$$

where $C[\ ]$ is an arbitrary context. We write $SPCF \vdash M = N$ if M and N are provably equal. ∎

Calculating with terms preserves their types. This property is important for many of the subsequent definitions though we will not explicitly mention it.[19]

Lemma 8.4 *If $A \vdash M : \tau$ and $SPCF \vdash M = N$ then $A \vdash N : \tau$.*

Proof. It is easy to check that the nine basic axioms have the following, slightly stronger property: If $A \vdash M : \tau$ and $M \longrightarrow N$ then $A \vdash N : \tau$. The property obviously holds for all rules except for (error), for which it is explicitly built in. The rest is a straightforward induction. ∎

An operational semantics for a language is a partial function from programs to answers, that is, in our case from closed SPCF terms of ground type to numerals and error elements. Based on the SPCF calculus we can define an evaluation function by saying that

[19]The property implies that e can only be the result of a program if it is a part of the original program text.

if a program is provably equal to a numeral or to an error element, the respective value is the program's result. With this, it is easy to confirm that the example programs from Section 3, $(\mathsf{catch}\ +_l)$ and $(\mathsf{catch}\ +_r)$, are equal to distinct outputs:

$$SPCF \vdash (\mathsf{catch}\ +_l) = \ulcorner 0 \urcorner$$

and

$$SPCF \vdash (\mathsf{catch}\ +_r) = \ulcorner 1 \urcorner.$$

However, Plotkin's analysis of the λ-calculus as a programming language suggests that such a definition is equivalent to the following, more algorithmic definition: to find the value of a program, it suffices to reduce the leftmost-outermost redex until the program is transformed into a value. Finding the leftmost-outermost redex in SPCF programs requires a partitioning of the program into an evaluation context and a redex. As for PCF, this partitioning process yields a unique pair of context and filler. If the filler is an error element, there may be several choices of splitting the program into an evaluation context and a redex. Otherwise, the pair is unique.

Lemma 8.5 (Unique Evaluation Context) *For any SPCF application M, there exists exactly one partitioning of M into an evaluation context $E[\ ]$ and a term R where R is either a free variable, Ω, $(\mathsf{sub1}\ 0)$, an error element, or a proper redex.*

Proof. The proof is an induction on the structure of M and proceeds by case analysis:

$M = (M_1\ M_2)$: If M_1 is a λ-abstraction, M is a redex and the empty context serves as the required evaluation context. Otherwise, there exists a unique partitioning of M_1 into some appropriate $E_1[\ ]$ and R by the inductive hypothesis. Take $M = E[R]$ as the partitioning where $E[\ ] = (E_1[\ ]\ M_2)$. By the inductive definition of the set of evaluation contexts, there cannot be any other partitioning.

$M = (f\ M_1)$ where $f \in \{\mathsf{add1}, \mathsf{sub1}, \mathsf{if0}\}$: If M_1 is a numeral, M is a redex or is $(\mathsf{sub1}\ \ulcorner 0 \urcorner)$. If $M_1 \in \mathbb{E}$, take $E = (f\ [\ ])$ and $R = M_1$. Otherwise, $M_1 = E_1[R]$ by the inductive hypothesis. As in the preceding case, $E[\ ] = (f\ E_1[\ ])$ and R are the desired pieces of M.

$M = (\mathsf{Y}\ M_1)$: M is a redex.

$M = (\mathsf{catch}\ (\lambda x_1 \ldots x_i.M_1))$ where $i \geq 0$ and M_1 is not a λ-abstraction: If M_1 is a numeral or an error element, the argument proceeds as in the preceding case. Otherwise, the inductive hypothesis guarantees that $M_1 = E_1[R]$. If R is a free variable among $x_1, \ldots, x_i$, then M is a redex of the right kind. If R is a redex, Ω, an error element, or some different free variable, the required partitioning for M consists of $E[\ ] = (\mathsf{catch}\ (\lambda x_1 \ldots x_i.E_1[\ ]))$ and R. ∎

As a consequence of the Unique Evaluation Context Lemma we can define evaluation for SPCF programs in the traditional way, namely, as leftmost-outermost reduction to a canonical form.

Definition 8.6 ($eval_{SPCF}$, $\mapsto$, *Values***)** The operational semantics of SPCF is the partial relation

$$eval_{SPCF} : \left\{ \begin{array}{rcl} Programs & \multimap\!\!\rightarrow & Values \\ M \;\; eval_{SPCF} \;\; w & \text{iff} & M \mapsto^* w \end{array} \right.$$

where

$$w \in Values = \mathbb{E} \cup \{\ulcorner n \urcorner \mid n \in \mathbb{N}\}$$

and

$$\frac{M \longrightarrow M'}{E[M] \mapsto E[M']} \, .$$

If $M \mapsto N$ we say M standard reduces to N; we call $\mapsto$ standard reduction. ∎

Due to (error), the relation $\mapsto$ is clearly not a function but a proper relation. For example, (add1 (add1 e)) $\mapsto$ (add1 e) and (add1 (add1 e)) $\mapsto$ e. Hence, it is necessary to prove that $eval_{SPCF}$ is a function. The proposition is an easy consequence of the Unique Evaluation Context Lemma.[20]

Proposition 8.7 *The relation $eval_{SPCF}$ is a partial function.*
Moreover, an SPCF program M evaluates either

1. *to a numeral,*
2. *to an error value (if $\mathbb{E} \neq \emptyset$),*
3. *to a stuck state,*
4. *or has an infinite standard reduction.*

Proof. The first part of the proposition is a straightforward consequence of the Unique Evaluation Context Lemma. If a program M is an application it either has a unique successor with respect to standard reduction or reduces to a unique error value. Hence, if $M \mapsto^* w$, then w is either a unique numeral or a unique error value.

The second part of the proposition follows from a Uniform Evaluation Lemma (see below). When applied to programs, which are closed and of ground type, the lemma says that for every SPCF program M one of the suggested alternatives holds. The possibilities are mutually exclusive due to the Unique Evaluation Context Lemma. ∎

Uniform evaluation for SPCF means that the reduction rules suffice to show that all SPCF terms M of ground type evaluate according to one of five possibilities: the four mentioned in the previous lemma and a stuck state of the shape $E[x]$ where *x* is free. The idea generalizes to all types.

[20]The proposition also follows from the Soundness Lemma (see 8.14) in the second subsection.

Definition 8.8 ($Eval_\tau$) The family of sets of SPCF phrases $Eval_\tau$ is defined by induction on types:

$$\begin{array}{rcll} Eval_o & = & \{M \mid & M \mapsto^* \ulcorner n \urcorner \text{ or} \\ & & & M \mapsto^* \mathsf{e} \text{ or} \\ & & & M \mapsto^* E[\Omega] \text{ or} \\ & & & M \mapsto^* E[x_i] \text{ or} \\ & & & \forall N.M \mapsto^* N \Longrightarrow \exists L.N \mapsto L\} \\ Eval_{\sigma\to\tau} & = & \{M \mid & M \mapsto^* \lambda x^\sigma.N \wedge N \in Eval_\tau \text{ or} \\ & & & M \mapsto^* E[\mathsf{e}] \text{ or} \\ & & & M \mapsto^* E[\Omega] \text{ or} \\ & & & M \mapsto^* E[x_i] \text{ or} \\ & & & \forall N.M \mapsto^* N \Longrightarrow \exists L.N \mapsto L\} \end{array}$$

∎

Lemma 8.9 (Uniform Evaluation) *For every SPCF term M of type τ, $M \in Eval_\tau$.*

Proof. The proof is an induction on the structure of τ. First assume $\tau = o$, $M \mapsto^* N$, and N is irreducible with respect to standard reduction. If $N \in Values$ one of the first two clauses of $Eval_o$ holds. Otherwise N is an application. In this case Lemma 8.5 says that $N = E[R]$ where $R \in Vars$, $R = \Omega$, $R \in \mathbb{E}$, or R is a proper redex. The first two cases are covered by clauses 3 and 4 of $Eval_o$. If $N = E[\mathsf{e}]$ then $N \mapsto^* \mathsf{e}$ because of its ground type. But this contradicts the assumption that N is irreducible, so it is impossible. The case where R is a redex leads to the same contradiction. Thus, M reduces according to one of the first four clauses or for every reachable term N there is a successor with respect to standard reduction.

Second assume $\tau = \sigma \to \nu$. The same analysis applies as in the base case except that irreducible terms are now λ-abstractions and evaluation contexts of type τ filled with $\mathsf{e} \in \mathbb{E}$. By inductive hypothesis, the bodies of λ abstractions are in $Eval_\nu$. ∎

The Uniform Evaluation Lemma implies an important lemma on the evaluation of phrases whose root is a catch application. The lemma is crucial for the treatment of catch in the following subsection. It requires the definition of a set of terms that suitably generalizes the set of terms with infinite standard reductions to higher types.

Definition 8.10 (Inf_τ) The family of sets of SPCF phrases Inf_τ is defined by induction on types:

$$\begin{array}{rcl} Inf_o & = & \{M \mid \forall N.M \mapsto^* N \Longrightarrow \exists L.N \mapsto L\} \\ Inf_{\sigma\to\tau} & = & \{M \mid M \mapsto^* \lambda x^\sigma.N \wedge N \in Inf_\tau, \forall N.M \mapsto^* N \Longrightarrow \exists L.N \mapsto L\} \end{array}$$

∎

The operator catch discovers when a term belongs to Inf.

Lemma 8.11 *Let M be a term of type τ. If $(\mathsf{catch}\ M) \in Inf_o$, then $M \in Inf_\tau$.*

Before closing the section, we collect four properties of the reduction and standard reduction relations that are crucial for proofs in the following subsection.

Proposition 8.12 *1. If $M \longrightarrow N$ then $M[L/x] \longrightarrow N[L/x]$.*

2. *If $M \mapsto M'$, then $E[M] \mapsto E[M']$.*
3. *If $M \mapsto N$ then $M[L/x] \mapsto N[L/x]$.*
4. *If $M \in \mathit{Inf}_\tau$ and its arity is a, then for all $L_1, \ldots, L_a$ and for all $N_1, \ldots, N_i$ and $x_1, \ldots, x_i$, $(M[N_1/x_1] \ldots [N_i/x_i]\ L_1 \ldots L_a) \in \mathit{Inf}_o$.*

Proof. The proof of the first part is straightforward for each notion of reduction. For (error) and (catch) the proof relies on Lemma 8.2, *i.e.*, that substitutions in an evaluation context do not affect its status as evaluation context.

For the second claim, if $M = E'[N]$, $N \longrightarrow N'$ and $M' = E'[N']$, then $E'' = E[E'[\ ]]$ is an evaluation context by Lemma 8.2 and $E[M] = E''[N]$. Hence, by definition of $\mapsto$, $E[M] \mapsto E''[N'] = E[M']$.

The third claim follows from the first one and Lemma 8.2.

Finally, the fourth claim requires a simple induction on the type τ. For $\tau = o$, $M \in \mathit{Inf}_o$ implies that M has an infinite reduction sequence. By part 3, all substitution instances of M also have an infinite reduction sequence and are thus in Inf_o. Thus assume $\tau = \sigma \rightarrow \nu$. Then either M itself has an infinite standard reduction, in which case the claim is again obviously true, or M standard reduces to some abstraction $\lambda x^\sigma.L$. In the latter case,

$$\begin{array}{rcl}(M[N_1/x_1] \ldots [N_i/x_i]\ L_1 \ldots L_a) & \mapsto^* & ((\lambda x^\sigma.L[N_1/x_1] \ldots [N_i/x_i])\ L_1 \ldots L_a) \\ & \mapsto & (L[N_1/x_1] \ldots [N_i/x_i][L_1/x^\sigma]\ L_2 \ldots L_a)\end{array}$$

again by parts 2 and 3 plus (β). By definition $L \in \mathit{Inf}_\nu$ and, hence by inductive hypothesis,

$$(L[N_1/x_1] \ldots [N_i/x_i][L_1/x^\sigma]\ L_2 \ldots L_a) \in \mathit{Inf}_o,$$

which proves the claim. ∎

8.2 Adequacy

To be useful, an operational semantics must faithfully realize the denotational semantics for programs. Technically, the operational semantics should map a program to a value if and only if the denotational semantics equates the program with the denotation of the value. Since this relationship is similar to the adequacy relationship between different semantics of a language, the corresponding theorem is called *Adequacy Theorem*.

Theorem 8.13 (Adequacy) *Let M be a program of SPCF$(\mathbb{E})$. Then the following equivalences hold:*

1. *$\mathcal{S}eq[\![\, M \,]\!] \perp = \{? \cdot n\}$ iff $\mathit{eval}_{SPCF}(M) = \ulcorner n \urcorner$;*
2. *$\mathcal{S}eq[\![\, M \,]\!] \perp = \{? \cdot \mathsf{e}\}$ iff $\mathit{eval}_{SPCF}(M) = \mathsf{e}$.*

(*$\mathcal{S}eq$ can be replaced by $\mathcal{SEQ}$ if $\mathbb{E} \neq \emptyset$.*)

The if-part of the Adequacy Theorem relies on a standard soundness proof for the reduction rules. Since the rules for catch differ from the rules of more traditional λ-calculi, we formulate the lemma and sketch its proof.

Lemma 8.14 (Soundness) *Let M and M' be terms that are typable with the same variable list A. Then*

1. *if $SPCF \vdash M = M'$, then $\mathcal{S}eq[\![A \vdash M]\!] = \mathcal{S}eq[\![A \vdash M']\!]$;*
2. *if $M \mapsto^* M'$ then $\mathcal{S}eq[\![A \vdash M]\!] = \mathcal{S}eq[\![A \vdash M']\!]$.*

Note: The type of M is not necessarily o in clause 2: the standard reduction relation applies to terms of higher type, too.

Proof. It is a routine exercise to check the basic reduction relations (add1), (sub1), (if0/0), (if0/1), and the respective inference rules. The validity of β- and fix follows from Lemma 3.6 and the results of Section 6. Next, the validation of (return) is simple. Clearly, for any appropriate list of typed variables A and matching environment tuple ρ,

$$\mathcal{S}eq[\![A \vdash \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.\ulcorner m \urcorner]\!] \star \rho = \{\langle ? \rangle \cdot \langle m \rangle\}.$$

Hence, by the semantics of catch,

$$\begin{array}{rl} & \mathcal{S}eq[\![A \vdash (\mathsf{catch}\ \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.\ulcorner m \urcorner)]\!] \star \rho \\ = & C_k \star (\mathcal{S}eq[\![A \vdash \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.\ulcorner m \urcorner]\!] \star \rho) \\ = & C_k \star (\{\langle ? \rangle \cdot \langle m \rangle\}) = \{? \cdot (m+k)\} \\ = & \mathcal{S}eq[\![A \vdash \ulcorner m+k \urcorner]\!] \star \rho \end{array}$$

The validity of the remaining rules, (catch) and (error), follows from a general claim about the denotations of terms of the shape $E[x]$:

> **Claim (i)**: For any evaluation context E and vector of variables $x_1^{\tau_1}, \ldots, x_k^{\tau_k}$, variable list A and matching environment vector ρ, if $i \leq k$, then
>
> $$\langle ? \rangle \cdot \langle i \rangle \in \mathcal{S}eq[\![A \vdash \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.E[x_i]]\!] \star \rho.$$

(Recall that, by convention, x_i occurs free in the hole of E.) Given the meaning of catch, it immediately implies the validity of (catch):

$$\begin{array}{rl} & \mathcal{S}eq[\![A \vdash (\mathsf{catch}\ \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.E[x_i])]\!] \star \rho \\ = & C_k \star (\mathcal{S}eq[\![A \vdash \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.E[x_i]]\!] \star \rho) \\ \sqsupseteq & C_k \star \{\langle ? \rangle \cdot \langle ?, i \rangle\} = \{? \cdot (i-1)\} \\ = & \mathcal{S}eq[\![A \vdash \ulcorner i-1 \urcorner]\!] \star \rho \end{array}$$

Together with the validity of (β), the claim also implies (error):

$$\begin{array}{rll} & \mathcal{S}eq[\![A \vdash E[\mathsf{e}]]\!] \star \rho = \mathcal{S}eq[\![A \vdash ((\lambda x^o.E[x^o])\ \mathsf{e})]\!] \star \rho & \text{by } (\beta) \\ \sqsupseteq & \{\langle ? \rangle \cdot \langle ?, 1 \rangle\} \star \{? \cdot \mathsf{e}\} = \{? \cdot \mathsf{e}\} & \text{by the semantics of application and the claim} \\ = & \mathcal{S}eq[\![A \vdash \mathsf{e}]\!] \star \rho. & \end{array}$$

Thus, we have reduced our task to the proof of Claim (**i**), which proceeds as follows:

$$
\begin{array}{lll}
 & & \mathcal{S}eq[\![\, A \vdash \lambda x_1^{\tau_1}, \ldots, x_k^{\tau_k}.E[x_i] \,]\!] \star \rho \\
 & = & \underbrace{\Lambda_t(\ldots\Lambda_t(}_{k} \mathcal{S}eq[\![\, A, x_1^{\tau_1}, \ldots, x_k^{\tau_k} \vdash E[x_i] \,]\!]) \ldots) \star \rho \\
(\mathbf{ii}) & \sqsupseteq & \underbrace{\Lambda_t(\ldots\Lambda_t(}_{k} \{\langle ? \rangle \cdot \langle\langle \ldots \langle\langle ? \rangle, 2\rangle, \underbrace{1\rangle, \ldots, 1\rangle}_{k-i}, 1\rangle, \ldots\}) \ldots) \star \rho \\
 & = & \{\langle\langle ? \rangle, 2\rangle \langle\langle ?, i\rangle, 2\rangle, \ldots\} \star \rho \\
 & = & \{\langle ? \rangle \cdot \langle ?, i\rangle, \ldots\}
\end{array}
$$

The step marked with (**ii**) requires a separate proof. For clarity, we formulate the claim explicitly:

> **Claim (ii)**: Let $E[\ ]$ be an evaluation context, and let $A = x_1^{\tau_1}, \ldots, x_k^{\tau_k}$ be a list of variables. Then
>
> $$\langle ? \rangle \cdot \langle\langle \ldots \langle\langle ? \rangle, 2\rangle, \underbrace{1\rangle, \ldots, 1\rangle}_{k-i}, 1\rangle \in \mathcal{S}eq[\![\, A \vdash E[x_i] \,]\!].$$

We prove Claim (**ii**) by induction on the structure of evaluation contexts:

1. If $E = [\ ]$ then $\mathcal{S}eq[\![\, A \vdash x_i \,]\!] = \pi^k_{k-i+1}$, and this projection clearly contains a path of the appropriate shape. Whence the conclusion immediately follows.

2. If $E = (f\ E_1)$, the result follows from the fact that all primitive functions f consult their first argument, which is $E_1[x_i]$: compare Definitions 7.1 and 7.2.

3. If $E = (E_1\ N)$, the conclusion follows from the fact that when an application is evaluated, the procedure, *i.e.*, $E_1[x_i]$, is consulted first.

4. If $E = \mathsf{catch}\ (\lambda y_1^{\sigma_1} \ldots y_m^{\sigma_m}.E_1)$, then by induction

   $$\langle ? \rangle \cdot \langle\langle \ldots \langle\langle ? \rangle, 2\rangle, \underbrace{1\rangle, \ldots, 1\rangle}_{k-i+m}, 1\rangle \in \mathcal{S}eq[\![\, A, y_1^{\sigma_1}, \ldots, y_m^{\sigma_m} \vdash E_1[x_i] \,]\!]$$

   hence

   $$
   \begin{array}{rcl}
   \langle ? \rangle \cdot \langle\langle \ldots \langle\langle ? \rangle, 2\rangle, \underbrace{1\rangle, \ldots, 1\rangle}_{k-i}, 1\rangle & = & \underbrace{\Lambda_t(\ldots\Lambda_t(}_{m} \langle ? \rangle \cdot \langle\langle \ldots \langle\langle ? \rangle, 2\rangle, \underbrace{1\rangle, \ldots, 1\rangle}_{k-i+m}, 1\rangle\ ) \ldots) \\
    & \in & \underbrace{\Lambda_t(\ldots\Lambda_t(}_{m} \mathcal{S}eq[\![\, A, y_1^{\sigma_1}, \ldots, y_m^{\sigma_m} \vdash E_1[x_i] \,]\!]\ ) \ldots) \\
    & = & \mathcal{S}eq[\![\, A \vdash \lambda y_1^{\sigma_1} \ldots y_m^{\sigma_m}.E_1[x_i] \,]\!]
   \end{array}
   $$

   and

   $$\langle ? \rangle \cdot \langle\langle \ldots \langle\langle ? \rangle, 2\rangle, \underbrace{1\rangle, \ldots, 1\rangle}_{k-i}, 1\rangle \in \mathcal{S}eq[\![\, A \vdash \mathsf{catch}\ (\lambda y_1^{\sigma_1} \ldots y_m^{\sigma_m}.E_1[x_i]) \,]\!]$$

   follows, since the sequentiality index of catch at $(\perp, \langle ? \rangle)$ is $\langle ? \rangle$. ∎

For the only-if part of the Adequacy Theorem, we use Berry *et al.*'s [6] proof method, which is a slight modification of Plotkin's computability method originally developed for the adequacy theorem for PCF [24]. We proceed in two steps. First, we prove an adequacy theorem for SPCF_1, which is SPCF without the Y operators. It has the same operational semantics as SPCF. Second, we lift the adequacy theorem for SPCF_1 by relating SPCF programs with Y to SPCF_1 programs with Y_n and by relating their evaluations.

To prove adequacy for SPCF_1, we first prove that all evaluation sequences are finite. The proof requires a generalization of this claim to closed phrases of *all* types.

Definition 8.15 ($Comp_\tau$) The family of sets of closed SPCF_1 phrases $Comp_\tau$ is defined by induction on types:

$$\begin{array}{rcl} Comp_o & = & \{M \mid M \mapsto^* \ulcorner n \urcorner, M \mapsto^* \mathsf{e}, M \mapsto^* E[\Omega], M \mapsto^* E[(\mathsf{sub1}\ 0)]\} \\ Comp_{\sigma\to\tau} & = & \{M \mid \forall N \in Comp_\sigma.(M\ N) \in Comp_\tau\}. \end{array}$$

We say $\vdash M : \sigma$ is normalizing if $M \in Comp_\sigma$. ∎

All SPCF_1 phrases are normalizing.

Lemma 8.16 (Normalization of SPCF_1) *For every SPCF_1 program M, $M \in Comp_o$.*

Proof. For notational convenience, F ranges over numerals, errors, and $E[\Omega]$ in this proof. The proof consists of three parts. The first claim shows that a phrase that standard reduces to a normalizing phrase is normalizing.

> **Claim 1**: Let M and M' be closed phrases of type τ. If $M \mapsto M'$ and $M' \in Comp_\tau$, then $M \in Comp_\tau$.

The proof is an induction on types:

1. Assume $\tau = o$. By assumption, $M' \mapsto^* F$ and $M \mapsto M'$. Hence, $M \mapsto^* F$.
2. Assume $\tau = \sigma \to \nu$. We must show $(M\ P) \in Comp_\nu$, for any $P \in Comp_\sigma$. By Lemma 8.12, $M \mapsto M'$ implies $(M\ P) \mapsto (M'\ P)$. Since $(M'\ P) \in Comp_\nu$ due to the assumptions, the inductive hypothesis implies that $(M\ P) \in Comp_\nu$, which is what we had to show. $\blacksquare_{Claim1}$

Second, we prove that substituting normalizing phrases for all free variables in an open phrase yields a normalizing phrase.

> **Claim 2**: If $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \sigma$, then $N_1 \in Comp_{\sigma_1}, \ldots, N_n \in Comp_{\sigma_n}$ implies $M[N_1/x_1, \ldots, N_n/x_n] \in Comp_\sigma$.

We use the notation $M[\vec{N}/\vec{x}]$ to abbreviate $M[N_1/x_1, \ldots, N_n/x_n]$. The proof procceds by induction on the structure of M:

1. $M = x_i$. Then $M[\vec{N}/\vec{x}] = N_i$, and the conclusion is one of the assumptions.

2. $M = \mathsf{e}$. Then $M[\vec{N}/\vec{x}] = \mathsf{e}$, and hence $M \in \mathit{Comp}_o$.

3. $M = \ulcorner n \urcorner$. This case is similar to the preceding case.

4. $M = \Omega$. If $\sigma_1 \to \ldots \to \sigma_m \to o$ is the type of M, we must prove $\Omega N_1 \ldots N_m \in \mathit{Comp}_o$. But this holds by definition since $\Omega N_1 \ldots N_m \mapsto^* \Omega$.

5. $M = (f\ M')$ where $f \neq \mathsf{catch}$. Assume $f = \mathsf{sub1}$. By the inductive hypothesis, $M'[\vec{N}/\vec{x}] \in \mathit{Comp}_o$, *i.e.*,

   (a) $M'[\vec{N}/\vec{x}] \mapsto^* \ulcorner 0 \urcorner$: then, $M \mapsto^* (\mathsf{sub1}\ \ulcorner 0 \urcorner)$;
   (b) $M'[\vec{N}/\vec{x}] \mapsto^* \ulcorner m+1 \urcorner$: then, $M \mapsto^* (\mathsf{sub1}\ \ulcorner m+1 \urcorner) \mapsto \ulcorner m \urcorner$;
   (c) $M'[\vec{N}/\vec{x}] \mapsto^* \mathsf{e}$: then, $M \mapsto^* (\mathsf{sub1}\ \mathsf{e}) \mapsto \mathsf{e}$
   (d) $M'[\vec{N}/\vec{x}] \mapsto^* E'[\Omega]$: then, $M \mapsto^* (\mathsf{sub1}\ E'[\Omega]) = E[\Omega]$ where $E[\ ] = (\mathsf{sub1}\ E'[\ ])$.

   In summary, for all possible cases, M is normalizing.

   The treatment of the cases for $f = \mathsf{add1}$ and $f = \mathsf{if0}$ is similar but easier.

6. $M = (\mathsf{Y}_n^\tau\ M')$. By induction hypothesis, $M'[\vec{N}/\vec{x}]$ is normalizing. Call this phrase N. The rest is proved by sub-induction on n. For $n = 0$, $M \mapsto \Omega^{(\tau\to\tau)}$. If $L_1, \ldots, L_a$ (where *a* is the arity of *M*) are normalizing expressions, then $(\Omega^{\tau\to\tau}\ L_1 \ldots L_a) \mapsto \Omega^o$, which proves that $\Omega^{\tau\to\tau}$ is normalizing. For $n > 0$, $(\mathsf{Y}_n\ N) \mapsto (N\ (\mathsf{Y}_{n-1}\ N))$. Since the sub-induction of this case implies that $\mathsf{Y}_{n-1}\ N \in \mathit{Comp}_\sigma$, $N\ (\mathsf{Y}_{n-1}\ N)) \in \mathit{Comp}_\sigma$ by definition of Comp_σ. By Claim 1, $M[\vec{N}/\vec{x}] \in \mathit{Comp}_\sigma$.

7. $M = \lambda x^\tau . M'$. Assume that $\sigma = \tau \to \nu$. Let N be a phrase in Comp_τ. Then $(M[\vec{N}/\vec{x}]\ N) \mapsto M'[N/x][\vec{N}/\vec{x}]$, which, by the inductive hypothesis, is normalizing. Hence, $(M[\vec{N}/\vec{x}]\ N) \in \mathit{Comp}_\nu$ for every $N \in \mathit{Comp}_\tau$, which is what we had to show.

8. $M = (M_1\ M_2)$. By the inductive hypothesis, $M_1[\vec{N}/\vec{x}]$ and $M_2[\vec{N}/\vec{x}]$ are normalizing. The rest follows by the definition of *Comp* at higher types.

9. $M = (\mathsf{catch}\ M')$. Set $N = M'[\vec{N}/\vec{x}]$, which is normalizing by inductive hypothesis. Assume *N* has type τ and arity *a*. Since $(\mathsf{catch}\ N)$ is a program, either $(\mathsf{catch}\ N) \mapsto^* F$ or $(\mathsf{catch}\ N) \in \mathit{Inf}_o$ by the Uniform Evaluation Lemma. We need to prove that the latter cannot happen. If it does, Lemma 8.11 implies that $N \in \mathit{Inf}_\tau$. By Proposition 8.12, for all $L_1, \ldots, L_a$, and in particular normalizing ones, $(N\ L_1\ \ldots\ L_a) \in \mathit{Inf}_o$. But this contradicts the normalizability of *N*. Hence, $M \in \mathit{Comp}_o$. $\blacksquare_{\mathit{Claim}\ 2}$

Finally, since closed phrases are just a special case of Claim 3, we have also shown that all SPCF_1 programs are normalizing. ∎

Based on the Normalization Lemma for SPCF_1, it is easy to prove an adequacy lemma.

Lemma 8.17 (Adequacy of $\mathbf{SPCF}_1$) *For all SPCF_1 programs M ($\vdash M : o$),*

1. *if $\mathcal{S}eq[\![\vdash M]\!] \star \bot = \{? \cdot m\}$ then $M \mapsto^* \ulcorner m \urcorner$;*

2. *if* $\mathcal{S}eq[\![\vdash M]\!] \star \bot = \{? \cdot \mathsf{e}\}$ *then* $M \mapsto^* \mathsf{e}$;

3. *if* $\mathcal{S}eq[\![\vdash M]\!] \star \bot = \{\}$ *then* $M \mapsto^* E[\Omega]$.

Proof. By the Normalization Lemma (8.16), the evaluation of M can only proceed in one of three possible ways:

1. $M \mapsto^* \ulcorner m \urcorner$: By Soundness, we get $\mathcal{S}eq[\![M]\!] \star \bot = \{? \cdot m\}$.

2. $M \mapsto^* \mathsf{e}$: By Soundness, $\mathcal{S}eq[\![M]\!] \star \bot = \mathsf{e}$ for any $\mathsf{e} \in \mathbb{E}$.

3. $M \mapsto^* E[\Omega]$: By Soundness, $\mathcal{S}eq[\![M]\!] \star \bot = \bot$.

Now if $\mathcal{S}eq[\![M]\!] \star \bot = m$, then $M \mapsto^* m$; anything else would contradict the preceding case analysis. The same argument works in the other two cases. ∎

Adequacy for full SPCF is a simple consequence of the preceding lemma.

Proof of Theorem 8.13. The right-to-left direction is a consequence of Lemma 8.14. The left-to-right direction follows from Lemma 8.17, the Adequacy Lemma for SPCF_1. The result is lifted to full SPCF as follows. The semantics of $(\mathsf{Y}\ M)$ is the least upper bound of the semantics of the expressions $(\mathsf{Y}_n\ M)$ for all $n \in \mathbb{N}$. Hence for any program M, if $\mathcal{S}eq[\![\vdash M]\!] \star \bot \neq \bot$, then by continuity there is some sufficiently large n, such that $\mathcal{S}eq[\![\vdash M]\!] = \mathcal{S}eq[\![\vdash N]\!]$, where N is the term obtained from M by replacing all the occurrences of Y in M by Y_n. By Lemma 8.17, N evaluates to its meaning. To obtain a reduction from M to the answer, it suffices to replace residuals of $(\mathsf{Y}_n\ L)$, which are terms of the form $(L\ (\ldots\ (L\ \Omega)\ldots))$, with $(\mathsf{Y}\ L)$. For details of this lengthy, but uniformative argument, we refer to Berry *et al.*'s report [6] or any of the modern textbooks on denotational semantics. This finishes the proof of the adequacy theorem. ∎

9 Generalizing Observable Sequentiality

Since PCF procedures have multiple sequentiality indices, PCF is clearly neither an observably sequential nor a manifestly sequential programming language (cmp. Definition 4.2). Consequently, the decision tree semantics for PCF is not fully abstract. However, the interesting question is which *practical* programming languages with non-local control operators are not manifestly sequential or observably sequential. At this point we know of one prominent example: sequential languages with *control delimiters* [12], also called *prompts*.

The task of a control delimiter is to mask any control operation that happens during the dynamic extent of some expression. Some typical examples in practical languages are Lisp's errset, Common Lisp's unwind-protect and ML's *wild-card* error-handler. Pragmatically, control delimiters are used to recover gracefully from unforeseen errors or to attach actions to non-local control operations. Languages with control delimiters are designed *not* to be manifestly sequential because the very task of control delimiters is to swallow all errors generated within their dynamic extent.

Consider the following example. Let PCF$'$ be PCF augmented by the control delimiter $\%$ and indexed error constants [30]. The defining equation for $\%$ is

$$[\![(\%\ M)]\!] = \begin{cases} [\![M]\!] & [\![M]\!] \in \mathbb{N}_\perp \\ i & [\![M]\!] = \mathsf{e}_i \in \mathbb{E} \end{cases} .$$

In PCF$'$, we can define an addition procedure that does not propagate errors:

$$+_{ei} = (\lambda xy\,.\,(+_l\ (\%\ x)\ (\%\ y))) = (\lambda xy\,.\,(+_r\ (\%\ x)\ (\%\ y))),$$

demonstrating that PCF$'$ is not a manifestly sequential language. A similar argument also shows that the extended language is not observably sequential. At this point, we do not know how to modify our construction to solve the full abstraction problem for PCF$'$ and other languages with control delimiters.

Acknowledgements. R. Cartwright and M. Felleisen thank Rama Kanneganti and Dorai Sitaram for helping hone their intuitions about manifest sequentiality, sequential functions, and for numerous discussions about SPCF. Steve Brookes pointed out a mistake in our original definition of syntactic sequentiality. John Gately and Trevor Jim discovered inconsistencies in the first versions of [7] and [10]. The authors gratefully acknowledge comments by Trevor Jim and Robert Harper on an early version of this manuscript. Finally, the authors want to thank the referees for their very careful readings.

A Sequential Data Structures vs. Concrete Data Structures

In this appendix, we connect the sequential data structures introduced in this paper with the concrete data structures of Berry-Curien. We define a category $\mathcal{S}eq(\mathbb{E})$ of observable sequential algorithms uniformly for any set of errors. We show that the definition of $\mathcal{S}eq(\mathbb{E})$ in Section 6 and the definition of the same category here are equivalent. We also show that the category $\mathcal{S}eq(\emptyset)$ is equivalent to Berry-Curien's category of sequential algorithms. In the process, we introduce filiform data structures, which are intermediate constructions in the gap between concrete data structures and sequential data structures.

A.1 Filiform Data Structures

We begin this subsection by recalling the definition of a concrete data structure. Then we show that *filiform* concrete data structures correspond through an equivalence of categories to the sequential data structures defined in this paper.

A filiform concrete data structure (*fcds* for short) is a quadruple $(C, V, E, \vdash)$ where C and V are collections of *cells* and *values*, respectively; E is a subset of $C \times V$; and $\vdash$ is a subset of $(E \times C) \cup C$. If $(c, v) \in E$, we say that (c, v) is an *event*, and that c is *filled* with v. We assume that each cell can be filled: for each c, there is a v such that $(c, v) \in E$. The set $\vdash$ is called the *enabling* relation. We write $(c_1, v_1) \vdash c$ to assert that $((c_1, v_1), c)$ is a member of $\vdash \cap (E \times C)$, and we say that event (c_1, v_1) enables cell c. We also say that c_1 precedes c. We write $\vdash c$ to assert that c is a member of $\vdash \cap C$. For $\vdash c$, we say that c is *initial* and has an empty enabling. We assume that the precedence relation is well-founded. An *fcds* is *canonical* if each cell has a unique enabling. A *state* of the *fcds* $(C, V, E, \vdash)$ is a subset x of E that is:

- *consistent*: if $(c, v_1) \in x$ and $(c, v_2) \in x$, then $v_1 = v_2$; and
- *safe*: if $(c, v) \in x$, then x contains an enabling of c.

We assume that *fcds*'s are *stable*, which means that given a state x and a cell c filled in x ($(c, v) \in x$ for some v), there is exactly one enabling of c in x. Canonical *fcds*'s are obviously stable. For non-canonical *fcds*'s, the stability condition must be proved for each domain construction, notably the exponent construction. In contrast, all sequential data structures are canonical and hence stable.

To help explain the connection between *fcds*'s with *sds*'s, we introduce the intermediate notion of a filiform data structure. The preceding definition of *fcds* is a restriction of the general definition of a (stable) concrete data structure (*cds*). In a general *cds*, an enabling consists of a finite collection of events. Thus the filiform case corresponds to restricting the maximum cardinality of enablings in a *cds* to 1. The category of *fcds*'s and sequential algorithms forms a full cartesian-closed subcategory of the category of stable *cds*'s and sequential algorithms [8: Theorem 2.6.7]. As a result, we are free to ignore the more general setting of stable *cds*'s and to take advantage of the special form of *fcds*'s. In the filiform case, the notions of "filling a cell" and of "enabling a cell" become dual in a loose sense. This fact can be emphasized by changing notation: we can write $c \prec (c, v)$ when $(c, v) \in E$, and $(c_1, v_1) \prec c$ when $(c_1, v_1) \vdash c$. These observations lead to the following definition.

Definition A.1 (*Filiform Data Structure*) A filiform data structure (*fds*) $(C, E, \prec)$ consists of a collection C of cells, a collection E of events, and a subset $\prec$ of $(C \times E) \cup (E \times C) \cup C$, called the *precedence* relation where:

- the precedence relation is well-founded; and
- each event e has a unique predecessor.

We write $\prec c$ to assert that c is a member of $\prec \cap\, C$. In this case, we say that c is *initial* and that it has an *empty enabling*.

The *states* of the *fds* are defined as the subsets x of E that are:

- *consistent*: if $c \prec e_1$, $c \prec e_2$, and e_1 and e_2 both belong to x, then $e_1 = e_2$; and
- *safe*: if $e \in x$, then either the predecessor c of e is initial, or there exists $e_1 \in x$ such that $e_1 \prec c \prec e$.

We assume that every *fds stable*: for any state x and any event $e \in x$, the predecessor c of e has at most one predecessor in x. As for *fcds*'s, we say that an *fds* is *canonical* if each cell has at most one predecessor. As before, canonicity implies stability. In addition, the well-foundedness of $\prec$ implies that for any state x and any $e \in x$, there exist $c_1, e_1 \ldots e_{n-1}, c_n$ such that $\prec c_1 \prec e_1 \prec \ldots \prec e_{n-1} \prec c_n \prec e$ and $e_1, \ldots, e_{n-1} \in x$. We call this a *proof* of e in x. ∎

The notions of *fcds* and of *fds* lead to equivalent categories. Although we have not yet defined these categories, the following "textual" correspondence should suffice to justify this claim:

- Given an *fcds* $(C, V, E, \vdash)$, construct the *fds* $(C, E, \prec)$, where $\prec c$ iff $\vdash c$; $e \prec c$ iff $e \vdash c$; and $c \prec e$ iff $e = (c, v)$ for some v.

- Given an *fds* $N = (C, E, \prec)$, construct the *fcds* $G(N) = (C, E, E', \vdash)$, E' is the set $\{(c, e) \,|\, e \in E, c \prec e\}$; $\vdash c$ iff $\prec c$; and $(c_1, e) \vdash c$ iff $e \prec c$.

These transformations induce order-isomorphisms between the respective sets of states and trees. If we compose the two transformations in either order, we obtain a structure isomorphic to the structure we started with. If we start with an *fcds* $(C, V, E, \vdash)$, we produce an *fcds* $(C, E, E', \vdash)$, whose states are obtained by replacing each (c, v) by $(c, (c, v))$. Similarly, if we start with an *fds* $(C, E, \prec)$, we obtain an *fds* $(C, E', \prec)$, whose states are obtained by replacing each e by (c, e), where c is the predecessor of e.

As for *sds*'s (and *fcds*'s), we write $\mathbb{D}(C, E, \prec)$ for the set of states of an *fds* $(C, E, \prec)$, ordered by inclusion.

We now connect *sds*'s and *fds*'s. Given an *sds* $\mathbf{M} = (A, D, P)$, we construct the *fds* $F(\mathbf{M}) = (Que, Res, \prec)$, where Que and Res are the sets of queries and responses of $\mathbf{M}$, respectively; $\prec p$ if and only if $|p| = 1$; and $p_1 \prec p_2$ if and only if $p_2 = p_1 \cdot u$ for some u. Notice that $F(\mathbf{M})$ is canonical. The following observations give us the ingredients for proving an equivalence of categories:

- The cpos $\mathbb{D}(\mathbf{M})$ and $\mathbb{D}(F(\mathbf{M}))$ coincide. This fact follows immediately from the observatins that prefix-closure implies safety, and glb-closure implies consistency.

- Any *fds* $(C, E, \prec)$ is isomorphic to the image of an *sds* (A, D, P) under F. We construct (A, D, P) as follows: let $A = C$, $D = E$, and let P be the set of all the finite precedence chains starting from an initial cell. Let the inverse mappings ϕ and ψ that define the one-to-one correspondence between $\mathbb{D}(C, E, \prec)$ and $\mathbb{D}(F(C, E, P))$ by defined as follows. The mapping ϕ replaces events e by their unique proof in x; ψ replaces every response with its terminal event. Notice the key role that stability plays in justifying the correspondence.

Since F maps *sds*'s to canonical *fds*'s, these observations also imply that *fds*'s and canonical *fds*'s define equivalent categories. Berry and Curien chose to work with a non-canonical presentation of the exponent of two canonical *cds*'s because this choice made it easier to define composition arrows in their category.

The preceding discussion of *fcds*'s and *sds*'s can easily be extended to include a given predefined set $\mathbb{E}$ of errors. We define states relative to $\mathbb{E}$ just as we did for trees in Section 5, and we call them observable states when $\mathbb{E}$ is not empty.

The composition of the transformations F from *sds*'s to *fds*'s, and G from *fds*'s to *fcds*'s, yields a mapping H that can be converted to an equivalence functor.

A.2 A Uniform Definition of Sequential Algorithms

In this subsection, we define categories of *sds*'s (and *fds*'s) with arrows that are (manifestly) sequential algorithms. If the set of errors $\mathbb{E} \neq \emptyset$, then the manifestly sequential algorithms correspond to functions and the methods used in Section 6 suffice. But if $\mathbb{E} = \emptyset$, we need to do more work to define a cartesian-closed category of either trees (using *sds*'s) or states (using *fds*'s). We will follow the approach taken by Berry-Curien to define an abstract algorithm between two *sds*'s, suitable for a simple definition of composition.

In the remainder of this section, we use the following notation. If x is a tree over an *sds* (with or without error values), we write $\mathit{Enabled}(x)$ for $\mathit{Answered}(x) \cup \mathit{Open}(x)$.

Definition A.2 (*Abstract algorithms***)** Let $\mathbf{M} = (A, D, P)$ and $\mathbf{M}' = (A', D', P')$ be two *sds*'s. Let Que, Res, Que', and Res' abbreviate $\mathit{Que}_{\mathbf{M}}$, $\mathit{Res}_{\mathbf{M}}$, $\mathit{Que}_{\mathbf{M}'}$, and $\mathit{Res}_{\mathbf{M}'}$, respectively. We let u range over $D' \cup \mathit{Que} \cup \mathbb{E}$, and w' range over $D' \cup \mathbb{E}$. An abstract algorithm from $\mathbf{M}$ to $\mathbf{M}'$ is a partial function f from $\mathbb{D}(\mathbf{M}) \times \mathit{Que}'$ to $D' \cup \mathit{Que} \cup \mathbb{E}$ satisfying the following axioms:

(A0) $f(x, q')$ defined $\Rightarrow\ \exists y \leq x$ (y finite and $f(y, q') = f(x, q')$),

(A1.1) $f(x, q') = d' \in D' \ \Rightarrow\ (q' \cdot d') \in P'$,

(A1.2) $f(x, q') = q \in \mathit{Que} \ \Rightarrow\ q \in \mathit{Open}(x)$,

(A2.1) $f(x, q') = w' \in D' \cup \mathbb{E},\ x \leq y \ \ \Rightarrow\ \ f(y, q') = w'$,

(A2.2) $f(x, q') = q,\ x \leq y$ and $q \in \mathit{Open}(y) \ \ \Rightarrow\ \ f(y, q') = q$,

(A2.3) $f(x, q') = q \ \Rightarrow\ f(x \cup \{q \cdot \mathsf{e}\}, q') = \mathsf{e}$, for any $\mathsf{e} \in \mathbb{E}$.

(A3.1) $f(x, q')$ defined $\ \Rightarrow\ \ q' \in \mathit{Enabled}(f * x)$,

(A3.2) $f(x, q')$ defined, $y \sqsubseteq x$ and $q' \in \mathit{Enabled}(f * y) \ \ \Rightarrow\ \ f(y, q')$ defined.

where

$$f * x = \{(q' \cdot w') \mid\ f(x, q') = w' \text{ for } w' \in D' \cup \mathbb{E}\}.$$

In the case where $\mathbb{E}$ is non-empty, we use the term *abstract algorithms with errors* instead of abstract algorithms.

Let $\mathbf{M}'' = (A'', D'', P'')$ be a third *sds*, and let g be an abstract algorithm from $\mathbf{M}'$ to $\mathbf{M}''$. The composition $h = g \circ f$ is defined as follows:

$$h(x, q'') \ = \ \begin{cases} q \in \mathit{Que}'' & \text{if } \ g(f * x, q'') = q' \text{ and } f(x, q') = q \\ w'' \in D'' \cup \mathbb{E} & \text{if } \ g(f * x, q'') = w'' \end{cases}$$

∎

The mapping $x \mapsto f * x$ defines a manifestly sequential function from $\mathbb{D}(\mathbf{M})$ to $\mathbb{D}(\mathbf{M}')$, which we call the *function* computed by f. When $\mathbb{E} \neq \emptyset$, the abstract algorithms are equivalent to the manifestly sequential functions defined in Section 6. But abstract algorithms support a uniform construction of $\mathcal{S}eq(\mathbb{E})$, independent of $\mathbb{E}$.

The proof that the composition is well-defined follows the corresponding proof for Berry and Curien's abstract algorithms [8: Proposition 2.6.1]. Consequently, we only check the axiom (A2.3) involving errors. If $h(x, q'') = q$, then $g(f * x, q'') = q'$ and $f(x, q') = q$. By (A2.3) applied to f and g, respectively, we have $f(x \cup \{q \cdot \mathsf{e}\}, q') = \mathsf{e}$ and $g((f * x) \cup$

$\{q' \cdot \mathsf{e}\}, q'') = \mathsf{e}$. From the first equation, we deduce that $(q' \cdot \mathsf{e}) \in f * (x \cup \{q \cdot \mathsf{e}\})$, implying $(f * x) \cup \{q' \cdot \mathsf{e}\} \leq f * (x \cup \{q \cdot \mathsf{e}\})$. The conclusion of the axiom follows from applying (A2.3) and (A2.1) to g.

We can now define a category of *sds*'s and (abstract) sequential algorithms.

Definition A.3 (*Seq*$(\mathbb{E})$) The category of (manifestly) sequential objects and algorithms over an error set $\mathbb{E}$ is defined as follows:

1. the collection of objects is $\{\mathbb{D}(\mathbf{M}) \,|\, \mathbf{M} \text{ is an } sds\}$, the set of (manifestly) sequential domains over *sds*'s relative to $\mathbb{E}$;
2. the collection of arrows between the objects $\mathbb{D}(\mathbf{M}_1)$ and $\mathbb{D}(\mathbf{M}_2)$ is $\mathbb{D}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$ (see Definition 6.21), the set of (observably) sequential algorithms relative to $\mathbb{E}$;
3. the composition operation for arrows is defined via the composition of trees when interpreted as abstract algorithms (see Definition A.2);
4. for each object the identity arrow is the identity algorithm.

∎

The main theorem of this appendix states that $Seq(\mathbb{E})$ is cartesian-closed, for any $\mathbb{E}$. This fact can be proved uniformly for any $\mathbb{E}$ by faithfully imitating the proofs of Curien [8]. In this subsection, we merely sketch the proof by partititioning it into two cases: $\mathbb{E} = \emptyset$ and $\mathbb{E} \neq \emptyset$. In each case, we show that the category *Seq*$(\mathbb{E})$ is equivalent to a category which we already know to be cartesian-closed.

Theorem A.4 *The category Seq*$(\mathbb{E})$ *is cartesian-closed, for all sets* $\mathbb{E}$ *of error values.*

Proof. If $\mathbb{E} = \emptyset$, we proceed as follows. The mapping H from *sds*'s to *fcds*'s defined in the previous subsection extends to a full and faithful functor from $Seq(\emptyset)$ to the category **ALGO** of stable *cds*'s and sequential algorithms, shown to be cartesian closed [8]. The functor H is actually into the full subcategory **fALGO** of *fcds*'s. This full subcategory is also cartesian-closed (the product and exponent of two *fcds*'s is filiform) [8: Theorem 2.6.7]. The arguments given in the last subsection can be easily turned into a formal proof that $Seq(\emptyset)$ is equivalent to **fALGO**. Hence, it is cartesian-closed.

If $\mathbb{E} \neq \emptyset$, it is straightforward to show that the abstract algorithms with errors are in one-to-one correspondence with the manifestly sequential functions. The bijection "forgets" an abstract algorithm f and takes "only" the function $x \mapsto f * x$ that it computes. The inverse bijection makes the sequentiality indices of a manifestly sequential function explicit. We omit the details. ∎

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 typés.* Thèse d'Etat, 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 J.-J. 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.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.
8. CURIEN, P.-L. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, London. 1986. Birkhäuser, Revised Edition, 1993.
9. CURIEN, P.-L. Sequentiality and full abstraction. In *Applications of Categories in Computer Science*, M.P. Fourman, P.T. Johnstone, and A.M. Pitts (Eds). Cambridge University Press, New York, 1992.
10. CURIEN, P.-L. Observable algorithms on concrete data structures. In *Proc. 7th Symposium on Logic in Computer Science*, 1992, 432–443.
11. CURIEN, P.-L. On the symmetry of sequentiality. In *Proc. Mathematical Foundations of Programming Semantics* 1993. Springer Lecture Notes in Computer Science, Berlin. To appear.
12. FELLEISEN, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 180–190.
13. FELLEISEN, M. AND D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, edited by M. Wirsing. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, 193–217.
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, 235–271.
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. JIM, T. AND A. MEYER. Full abstraction and the context lemma. In *Proc. International Conference on Theoretical Aspects of Computer Software (TACS)*. Lecture Notes in Computer Science 526. Springer Verlag, Berlin Heidelberg 1991. 131–151.

17. JUNG, A., AND A. STOUGHTON. Studying the fully abstract model of PCF within its continuous function model. In *Proc. Conference on Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science 664. Springer Verlag, Berlin Heidelberg 1993.

18. KAHN, G. AND G. PLOTKIN. Domaines Concrets. IRIA Report 336. 1978. English translation: Concrete domains, to appear in C. Boehm Festschrift, special volume of *Theoretical Computer Science*.

19. KANNEGANTI, R., R. CARTWRIGHT, AND M. FELLEISEN. SPCF: its model, calculus, and computational power. In *Proc. REX Workshop on Semantics and Concurrency*. Lecture Notes in Computer Science 666. Springer Verlag, Berlin Heidelberg 1993. 318–347.

20. LAMARCHE, M. Sequentiality, games and linear logic. Unpublished manuscript. Ecole Normale Supérieure, 1992.

21. MEYER, A. R. AND K. SIEBER. Towards a fully abstract semantics for local variables. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 191–203.

22. MILNER, R. Fully abstract models of typed λ-calculi. *Theor. Comput. Sci.* **4**, 1977, 1–22.

23. MULMULEY, K. *Full Abstraction and Semantic Equivalence*. MIT Press, Cambridge, Massachusetts, 1986.

24. PLOTKIN, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.

25. PLOTKIN, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.* **1**, 1975, 125–159.

26. SAZONOV, V.Y. Expressibility of functions in D.Scott's LCF language. *Algebra i Logika* **15**(3), 1976, 308–330.

27. SCOTT, D. S. Domains for denotational semantics. In *Proc. International Conference on Automata, Languages, and Programming*, Lecture Notes in Mathematics 140, Springer Verlag, Berlin, 1982, 577–613.

28. SCOTT, D.S. *Lectures on a Mathematical Theory of Computation.* Techn. Monograph PRG-19, Oxford University Computing Laboratory, Programming Research Group, 1981.

29. SIEBER, K. Relating full abstraction results for different programming languages. In *10th Conference on Foundations of Software Engineering and Theoretical Computer Science*. K.V. Nori and C.E. Veni Madhavan, Eds. Lecture Notes in Computer Science 472. Springer Verlag, Berlin, 1990, 373–387.

30. 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. Corrections posted to the "continuations" email list on September 22, 1992 (also available from titan.cs.rice.edu via anonymous ftp in public/languages as lfp90-sf-correction.{ps,dvi}.Z).

31. STEELE, G.L., JR. AND G.J. SUSSMAN. The revised report on Scheme, a dialect of Lisp. Memo 452, MIT AI-Lab, 1978.

32. STOUGHTON, A. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman, London. 1986.

33. VUILLEMIN, J. Proof techniques for recursive programs. IRIA Report. 1973.

Notation

Notation	Description	Section
Sets		
$\mathbb{N}$	the set of natural numbers	2
$A \setminus B$	set minus	2
$A \uplus B$	tagged, disjoint union	2
$(\underline{\lambda} x : A. \cdots x \cdots)$	mathematical function from A to B	2
$\langle x, y \rangle$	cartesian pair of x and y	2
Domains		
$(D, \sqsubseteq)$	partial order, approximation relation	2
$\sqsubset$	strict partial order (below)	2
$\sqcup$	least upper bound	2
$x \uparrow y$	bounded (consistent) elements x and y	2
$\sqcap$	greatest lower bound	2
$\perp$	least element	2
$\mathbb{N}_\perp$	flat domain of natural numbers	2
Paths		
Σ^*	set of paths over alphabet Σ	2
ϵ	empty path	2
$a \cdot p$	*cons*ing a onto path p	2
$p \cdot a$	*cons*ing a onto the end of p	2
$q \cdot p$	appending q onto p	2
$p \,@\, n$	nth element in path p	2
$(\Phi, \Psi)^*$	set of alternating paths	2
$\circ$	composition of arrows	2
Categories		
id^A	identity arrow for object A	2
1	terminal object	2
1^A	unique arrow from object A to 1	2
$A \times B$	cartesian product of A and B	2
π_i	cartesian projection to the ith component	2
π_i^n	ith cartesian projection for n fold product	2
$A \Rightarrow B$	exponent of A and B	2
Λ, Λ^{-1}	curry function and inverse	2
App	application arrow	2

Notation	Description	Section
PCF syntax		
o	PCF ground type (number)	3
$\tau \to \sigma$	PCF procedure type	3
x	PCF variable	3
x^τ	typed PCF variable	3
$\lambda x.M$	PCF procedure	3
$(M\ N)$	PCF application	3
$(f\ M)$	PCF primitive application	3
$\ulcorner n \urcorner$	PCF numeral	3
$\mathsf{add1}$	PCF successor primitive	3
$\mathsf{sub1}$	PCF predecessor primitive	3
$\mathsf{if0}$	PCF zero test primitive	3
Y	PCF fixed-point operator	3
$C[\ ]$	PCF single hole context	3
$C[\ ]_1 \ldots [\ ]_n$	PCF multiple hole context	3
$A \vdash M : \tau$	PCF type judgement	3
Ω	Syntactic representation of $\perp$	3
Y^n	nth approximation of Y	3
Semantics		
$\mathcal{C}$	meta-variable for PCF semantics	3
$\mathcal{D}om$	continuous functions semantics of PCF	3
$\sqsubseteq_{\mathcal{C}}$	denotational approximation for terms, relative to a semantics $\mathcal{C}$	3
$\equiv_{\mathcal{C}}$	denotational equivalence relation for terms, relative to a semantics $\mathcal{C}$	3
$\sqsubseteq\!\!\!\!\sim$	operational approximation	3
$\simeq$	operational equivalence	3
$+_l$	left addition	3
$+_r$	right addition	3
SPCF		
$\mathbb{E}$	SPCF set of error values	4
e	element of $\mathbb{E}$	4
catch	SPCF catch operation	4
$L_{\mathbb{E}}$	set of error expressions in PCF-like language	4
E	element of $L_{\mathbb{E}}$	4

Notation	Description	Section
***sds*s**		
$\mathbf{M} = (A, D, P)$	sequential data structure	5
$Res_{\mathbf{M}}$	responses of *sds* $\mathbf{M}$	5
$Que_{\mathbf{M}}$	queries of *sds* $\mathbf{M}$	5
$\mathbf{N}$	*sds* of natural numbers	5
$\mathbf{T}$	*sds* of booleans	5
$\mathbf{N}^4 \to \mathbf{N}$	*sds* for algorithms from N^4 to N	5
$\mathbb{D}(\mathbf{M})$	domain over *sds* $\mathbf{M}$	5
$\mathbb{D}_0(\mathbf{M})$	finite elements in $\mathbb{D}(\mathbf{M})$	5
$\mathbb{N}^e_\perp$	flat domain of natural numbers and errors	5
$Open(x)$	open queries of element x	5
$Answered(x)$	answered queries in element x	5
$\mathbb{F}(\mathbf{M}_1 \Rightarrow \mathbf{M}_2)$	(observably) sequential functions: $\mathbb{D}(\mathbf{M}_1) \longrightarrow \mathbb{D}(\mathbf{M}_2)$	5
$\mathsf{si}_f(x, q')$	sequentiality index of f at x and q'	5
Function Trees		
$\mathcal{SEQ}(\mathbb{E})$	category of (observably) sequential domains and functions	6
$\mathcal{S}eq(\mathbb{E})$	category of (observably) sequential domains and functions-as-trees	6
$\pi_i^\times$	projection for cartesian products in $\mathcal{SEQ}(\mathbb{E})$	6.1
inj_i	injection for cartesian products in $\mathcal{SEQ}(\mathbb{E})$	6.1
$\\|s\\|$	domain element determined by s	6.2.1
$\pi_1^\Rightarrow$	projection to input *sds* of exponent	6.2.1
$\pi_2^\Rightarrow$	projection to output *sds* of exponent	6.2.1
$\star$	application operation	6.2.1
Fun	isomorphism from exponent to function space	6.2.1
Tree	inverse of Fun	6.2.1
$\mathsf{Path}_f(x, p)$	path for simulating f on x with $p \in f(x)$	6.2.1
$\Lambda_t, \Lambda_t^{-1}$	curry function and inverse for trees	6.2.2
SPCF Semantics		
$\langle ? \rangle$	initial address for $\mathbb{D}(\mathbf{M}_1 \Rightarrow \ldots \mathbf{M}_k \Rightarrow N)$	7
$\langle n \rangle$	initial datum behind $\langle ? \rangle$	7
$\langle p, i \rangle$	path in ith part of $\mathbb{D}(\mathbf{M}_1 \Rightarrow \ldots \mathbf{M}_k \Rightarrow N)$	7
π_i	$= \pi_1^\Rightarrow(\underbrace{\pi_2^\Rightarrow(\ldots \pi_2^\Rightarrow}_{i-1}(p)\ldots)$ for $\mathbb{D}(\mathbf{M}_1 \Rightarrow \ldots \mathbf{M}_k \Rightarrow N)$	7
A	denotation of $\mathsf{add1}$	7
S	denotation of $\mathsf{sub1}$	7
I	denotation of $\mathsf{if0}$	7
C	denotation of catch	7

Notation	Description	Section
Operational Semantics		
$E[\ ]$	evaluation context	8
$M \longrightarrow N$	SPCF reduction relation	8
$M = N$	SPCF calculus	8
$M \mapsto N$	SPCF standard reduction	8
$eval_{SPCF}$	SPCF evaluator for	8
$Eval_\tau$	uniform evaluation set	8
Inf_τ	set of terms with infinite evaluation	8
$Comp_\tau$	set of tersm with finite evaluation	8
Appendix		
$(C, V, E, \vdash)$	concrete data structure	A.1
$(C, E, \prec)$	filiform data structure	A.1
$*$	application of abstract algorithms	A.2

Contents