Observable Sequentiality and Full Abstraction

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

Rice Technical Report CS 91-167

A preliminary version of this technical report appeared in the
Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages,
January 19–22, 1992, Albuquerque, New Mexico.

Copyright ©1992 by Robert Cartwright and Matthias Felleisen

1 Full Abstraction and Sequentiality **1**
1.1 Summary of Previous Work 2
1.2 Summary of Results 3

2 PCF **4**
2.1 Formal Semantics 5
2.2 Observational Equivalence, Sequentiality 8

3 Observing Sequentiality **9**
3.1 Using errors, programmers can observe the order of evaluation. 10
3.2 Using control operators, programs can observe the order of evaluation. . . . 10
3.3 Procedure denotations contain more computational structure than ordinary function graphs. 11
3.4 Higher-order procedures explore trees sequentially. 13

4 The Tree Model **15**
4.1 Tree Domains 15
4.2 Application 24
4.3 Assigning Meaning to Phrases 31
4.4 Operational Properties of the Model 37

5 The Full Abstraction Theorem **41**

6 Observable Sequentiality in SPCF **48**

7 Generalizing Observable Sequentiality **50**

A K and S **51**
A.1 The K Combinator 51
A.2 The S Combinator 52

B Equations for catch and error **63**

Observable Sequentiality and Full Abstraction

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

August 13, 1992

Abstract

One of the major challenges in denotational semantics is the construction of fully abstract models for *sequential* programming languages. For the past fifteen years, research on this problem has focused on developing models 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 in procedures. Since we believe that such facilities are crucial for understanding the nature of sequential computation, this paper focuses on a sequential extension of PCF (called SPCF) that includes two classes of control operators: error generators and escape handlers. These new control operators enable us to construct a fully abstract model for SPCF that interprets higher types as sets of *error-sensitive* functions instead of *continuous* functions. The error-sensitive functions form a Scott domain that is isomorphic to a domain of decision trees.

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 denote the same element in the model. The second relation, *observational equivalence*, equates two phrases if and only if they have the same "observable 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.

These two equivalence relations are different in general. Denotational equivalence implies observational equivalence because denotational models are compositional. However, the converse rarely holds for conventional models of *practical* programming languages; some observationally equivalent phrases inevitably have different meanings. A semantics in which denotational equivalence and observational equivalence coincide is said to be *fully abstract*.

Observational equivalence is the fundamental constraint governing program optimization. Since the users of a program are primarily interested in its input-output behavior, a

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

compiler can optimize a program by interchanging observationally equivalent phrases. If a denotational semantics is fully abstract, then observational equivalence reduces to denotational equivalence, which can be analyzed using algebraic methods. Consequently, one of the primary goals of research in denotational semantics has been the construction of fully abstract models for practical languages.

The following example shows why conventional continuous function models are not fully abstract for higher-order sequential languages. Consider the following family of procedures[1] p_i defined in a sequential, call-by-name functional language L:

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

In this equation, Ω denotes any divergent expression and i denotes an arbitrary natural number. It is easy to show by an exhaustive case analysis that the procedures p_i diverge for all inputs because the only possible inputs f are *sequential* procedures. Hence, p_i is observationally equivalent to the procedure p defined by $p(f) = \Omega$.

On the other hand, the conventional model 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. If we apply p_i to *parallel-and*, the computation produces the answer i instead of $\perp$ because the divergent subcomputations are ignored by *parallel-and*. Consequently, the conventional model for such functional languages 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 [15, 22]. 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 [22]. Indeed, all commonly used deterministic languages are *sequential*. In informal terms, a language is sequential if it can be implemented without time-slicing among multiple threads of control. Practical languages eschew parallel operations like *parallel-and* because they are painful to implement and encourage hideously inefficient programming. Even *deterministic* languages for *parallel* machines like C* [19] are sequential.

1.1 Summary of Previous Work

Milner [16] and Plotkin [18] were the first researchers to study the full abstraction problem for sequential languages. They focused on constructing fully abstract models for PCF, a call-by-name functional language based on the typed λ-calculus. The preceding example shows that the continuous function model for PCF is not fully abstract. Milner and Plotkin developed different strategies for eliminating the discrepancy between the continuous function model and the observable behavior of PCF.

Milner eliminated parallel functions from the model by constructing a syntactic model in which the elements are equivalence classes of observationally equivalent terms. Although Milner's model is fully abstract, it is not generally regarded as a "denotational" model

[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 in models.

because it does not identify any algebraic structure within programs. Plotkin extended PCF by adding *parallel* deterministic operations, eliminating the discrepancy between the continuous function model and the procedures definable in the language. Unfortunately, neither Milner's nor Plotkin's result showed how to construct fully abstract denotational models for *sequential* languages.

In a subsequent investigation of sequential languages, Berry and Curien [3, 4, 6, 8] constructed models for PCF with more restrictive domains of procedure denotations. Berry eliminated many parallel functions from these domains by forcing functions to be *stable*. This construction eliminated some of the spurious distinctions between phrases in the conventional model, but it introduced some new ones.[2] To address this problem, Berry and Curien [4, 5] proposed interpreting procedures as *sequential algorithms over concrete domains* [14]. Roughly speaking, a concrete domain is a domain that is isomorphic to a domain consisting of potentially infinite trees. A sequential algorithm is a function *plus* a strategy for evaluating its arguments. While this approach eliminates all parallel functions, the resulting model is not extensional because it contains different sequential algorithms with exactly the same behavior under application. In addition, PCF cannot express all of the observations that characterize sequential algorithms, such as the order of argument evaluation. As a result, the sequential algorithm model for PCF is not fully abstract.

Recently, Mulmuley [17] generalized Milner's work by showing how to construct a fully abstract model for PCF as a quotient of a conventional model based on *lattices* instead of *cpos*. Mulmuley defined a retraction on the conventional model that equates all parallel functions with the "overdefined" element top ($\top$). However, like Milner's original fully abstract model, Mulmuley's model is "syntactic" in flavor because it relies on a quotient construction based on observational equivalence. For more details on the history of the full abstraction problem for sequential languages, we refer the reader to two extensive surveys [6, 25].

1.2 Summary of Results

Fifteen years after Milner's and Plotkin's original work, the fundamental question still remains:

> *Are there fully abstract denotational models for sequential programming languages?*

In this paper, we answer the question affirmatively by showing how to construct a fully abstract denotational model for an *observably sequential* functional programming language that is a simple sequential extension of PCF. The construction relies on two closely related insights:

1. In *practical* languages, a programmer can use run-time errors to observe the order in which a procedure evaluates its arguments. Similarly, many practical languages

[2] This observation is due to T. Jim and A. Meyer [13]. In the stable function model, two procedures corresponding to the same continuous function can denote *inconsistent* stable functions. The procedures $+_l$ and $+_r$ defined in Section 2 have this property. As a result, there is a higher order stable function $+_l?$ that maps $+_l$ to 0 and $+_r$ to 1. In PCF, $+_l?$ is not definable, but it is easy to define a family of "recognizers" $\{q_i \mid i \in \mathbb{N}\}$ for $+_l?$ (analogous to the procedures p_i described earlier) that diverge on all definable inputs but map $+_l?$ to i.

permit programs to determine and exploit their sequential evaluation order by using explicit control operators such as exception generators and handlers.

2. Continuous functions can be identified with decision trees if they are constructed by composing *error-sensitive* operations. In informal terms, an operation is *error-sensitive* if it propagates any errors that it encounters in evaluating its arguments.

The remainder of this paper is organized as follows. After presenting the syntax and informal semantics of PCF in Section 2, we compare PCF to realistic functional languages in Section 3. Unlike nearly all practical languages, 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. If we extend PCF to include these facilities, then we can construct a fully abstract model using error-sensitive functions represented by decision trees. We define such a model in Section 4 and show that it is extensional. In Section 5, we prove the full abstraction theorem. In Section 6, we rigorously define the properties of SPCF that make the construction of a fully abstract model possible: *error-sensitivity* and *observable sequentiality*. Finally, in the last section we suggest how our model of SPCF can be adapted to other functional languages that are *error-sensitive* and *observably sequential*.

2 PCF

PCF is a simple functional language based on the typed λ-calculus with numerals, procedures to increment and decrement numbers, a conditional procedure, and a family of fixed point operators. The set *Type* of PCF types consists of a single ground type (o), denoting the set of non-negative integers, and an infinite collection of procedure types ($\sigma \to \tau$). Each procedure type $\sigma \to \tau$ consists of the set of all procedure denotations that can be applied to values of type σ to produce values of type τ. The set *Type* is formally defined by the recursive rule:

$$\mathit{Type} ::= o \mid (\mathit{Type} \to \mathit{Type}).$$

This rule is equivalent to the infinite set of rules

$$\mathit{Type} ::= o \mid (\underbrace{\mathit{Type} \to \ldots \to \mathit{Type}}_{n} \to o) \qquad \text{for all } n > 0,$$

which shows that procedures can be interpreted as maps from n-ary tuples to ground results. The second view also emphasizes the fact that all procedures have fixed arity.

A PCF phrase M is either a constant $f \in \mathcal{F}$ (including numerals $\ulcorner n \urcorner$, $n \geq 0$), a typed variable $x^\sigma \in \mathcal{V}$, a λ-abstraction, or an application:

$$\begin{array}{rcl} M & ::= & \mathcal{F} \mid \mathcal{V} \mid (\lambda \mathcal{V} \,.\, M) \mid (M\ M) \\ \mathcal{F} & ::= & \mathsf{add1} \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{Y}^\sigma \mid \ulcorner n \urcorner \\ \mathcal{V} & ::= & x^\sigma \mid y^\sigma \mid \ldots \end{array}$$

that conforms to the type constraints of the typed λ-calculus. In the typed λ-calculus, every phrase M has a unique type τ. Each variable x^σ has type σ. Similarly, each constant $fin\mathcal{F}$

in the language has a designated type τ_f. Within a well-formed 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.

The constants of PCF have the following types. Each numeral $\ulcorner n \urcorner$ has type o. The procedural constants $\mathsf{add1}$ and $\mathsf{sub1}$ have type $o \to o$, $\mathsf{if0}$ has type $o \to o \to o \to o$, and Y^σ has type $(\sigma \to \sigma) \to \sigma$.

The *programs* of PCF are the closed PCF phrases of ground type. A *syntactic context* C is a phrase with "holes" $[\ ]$ in place of some subexpressions. The phrase $C[M_1, \ldots, M_n]$ is the result of filling the holes of C with the corresponding expressions $M_1, \ldots, M_n$, possibly capturing free variables in $M_1, \ldots, M_n$ in the process. We will denote the set of free variables in the phrase M by $FV(M)$.

In PCF, numerals denote numbers, λ-abstractions denote call-by-name procedures, juxtapositions of the form $(Y\ M)$ denote the least fixed points of the functions corresponding to M, other juxtapositions denote procedure applications, and the procedural constants $\mathsf{add1}$, $\mathsf{sub1}$, $\mathsf{if0}$, and Y^σ have their usual meanings. The procedures $\mathsf{add1}$ and $\mathsf{sub1}$, respectively, increment and decrement their integer inputs; 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 for procedures of type $\sigma \to \sigma$. In PCF program text, we will use the constant Ω^σ to abbreviate the phrase $(\mathsf{Y}^\sigma\ (\lambda x^\sigma\,.\,x^\sigma))$, the prototypical divergent expression of type σ.

The following equations define two different versions of the binary addition procedure in PCF:

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

The first version $+_l$ recurs on the first argument x; the second version recurs on the second argument y. In the conventional continuous denotational semantics for PCF described below, these two procedures denote exactly the same function. We will use these two procedures in Section 3 when we discuss practical extensions to PCF.

2.1 Formal Semantics

To define the semantics of PCF more rigorously, we introduce the concept of a *semantic definition.*

Definition 2.1. (*Semantic Definition*) Let L be a language based on the typed λ-calculus. A *semantic definition* for L is a function $\mathcal{P}$ mapping the programs of L to meanings in a set $\mathbb{A}$ of *answers.* ∎

In the literature, languages are usually defined "operationally" by a collection of syntactic rules that reduce a program P to an answer a. A semantic definition only assigns meaning to programs; it says nothing about the meaning of program phrases that are not programs. In practice, programmers need to reason about the interaction of program components like procedures. To facilitate this process, language designers often develop a more

comprehensive definition that assigns meanings to all program phrases. Such an interpretation is called a *model*. Of course, a model for a language must agree with the meanings assigned to programs by the semantic definition. A model that assigns the same meaning to programs as the semantic definition is said to be *adequate*.

Since a model assigns meanings to programs (as well as incomplete program phrases), it can also serve as a semantic definition. The advantage of this approach is simplicity: it eliminates the need to present a separate operational definition and prove that the model is *adequate* with respect to the operational definition. We will use the "model-based" approach to language definition in this paper.

In defining a model for a language based on the typed λ-calculus, it is convenient to transform programs to *combinatory* form, which eliminates *bound* variables and λ-abstractions at the cost of adding the constants $\mathsf{S}^{\sigma,\tau,\upsilon}$, $\mathsf{K}^{\sigma,\tau}$, and I^{σ} and the binary function symbols $\mathsf{apply}^{\sigma,\tau}$ for all types σ, τ, and υ. The new constants S, K, and I are called *combinators*. The combinator $\mathsf{S}^{\sigma,\tau,\upsilon}$ has type $(\sigma \to \tau \to \upsilon) \to (\sigma \to \tau) \to \sigma \to \upsilon$, K has type $\sigma \to \tau \to \sigma$, and I has type $\sigma \to \sigma$. The function symbol $\mathsf{apply}^{\sigma,\tau}$ accepts inputs in the cross product of types $(\sigma \to \tau) \times \sigma$ and produces outputs of type τ.

In the transformation process, all procedure applications are made explicit because some models do not interpret procedures as functions. The syntactic forms produced by this translation are called combinatory terms. They are defined as follows.

Definition 2.2. (*Combinatory Term*) Let L be a language based on the typed λ-calculus with variables $\mathcal{V}$ and constants $\mathcal{F}$. Let the sets of symbols $\mathcal{O}$ and $\mathcal{A}$ be defined by the equations:

$$\begin{aligned} \mathcal{O} &= \{\mathsf{S}^{\sigma,\tau,\upsilon} \mid \sigma,\tau,\upsilon \in \mathit{Type}\} \cup \{\mathsf{K}^{\sigma,\tau} \mid \sigma,\tau \in \mathit{Type}\} \cup \{\mathsf{I}^{\sigma} \mid \tau \in \mathit{Type}\} \\ \mathcal{A} &= \{\mathsf{apply}^{\sigma,\tau} \mid \sigma,\tau \in \mathit{Type}\}\,. \end{aligned}$$

A *combinatory term* in L is an expression E that conforms to the following inductive definition (expressed in BNF):

$$E \quad ::= \quad \mathcal{V} \mid \mathcal{F} \mid \mathcal{O} \mid \mathcal{A}(E,E)$$

and the type constraints of typed λ-calculus. ∎

The transformation of PCF to combinatory form is given in Figure 1 as part of the definition of the conventional model for PCF.

We rigorously define the notion of a model and the semantic definition induced by a model as follows.

Definition 2.3. (*Denotational Model*[3]) Let L be a language conforming to the syntax of the typed λ-calculus with variables $\mathcal{V}$ and constants $\mathcal{F}$. A *denotational model* $\mathcal{M}$ for L is

[3] In the logic literature, the word *structure* is often used instead of *model* to identify an interpretation function $\mathcal{M}$. In logic, the word *model* is usually reserved for structures that satisfy a particular set of axioms Γ. The same distinction between the words *structure* and *model* can be made in the context of program semantics: the operational semantics for a language can be interpreted as a collection of axioms about programs. But the word *structure* is a poor choice of words in the context of programming languages where *structure* is already overused. Consequently, the word *model* is commonly used in semantics to refer both to arbitrary structures and structures that satisfy a given set of axioms Γ. In this paper, no ambiguity arises because we never define an operational semantics for our programming languages PCF and SPCF.

a function interpreting each type τ, each constant in $\mathcal{F} \cup \mathcal{O}$, and each function symbol in the set $\mathcal{A}$ (of **apply** symbols). $\mathcal{M}$ maps each type τ to a Scott domain $\mathcal{M}^\tau$, each constant c of type τ to an element in $\mathcal{M}^\tau$, and each function symbol $\mathsf{apply}^{\sigma,\tau} \in \mathcal{A}$ to a function $\mathcal{M}[\![\mathsf{apply}^{\sigma,\tau}]\!] : \mathcal{M}^{\sigma\to\tau} \times \mathcal{M}^\sigma \longrightarrow \mathcal{M}^\tau$. ∎

A denotational model $\mathcal{M}$ for L determines a meaning for every closed program phrase in L.

Definition 2.4. (*Semantics of Closed Phrases*) The *meaning* of a closed phrase N *determined by the model* $\mathcal{M}$ (denoted $\mathcal{M}[\![N]\!]$) is the *algebraic* meaning that $\mathcal{M}$ assigns to the combinatory term $[N]_{CL}$ corresponding to N. The *algebraic* meaning of a constant c is simply $\mathcal{M}[\![c]\!]$. The *algebraic* meaning of a composite term $\mathsf{apply}^{\sigma,\tau}(N_1, N_2)$ is $\mathcal{M}[\![\mathsf{apply}^{\sigma,\tau}]\!](\mathcal{M}[\![N_1]\!], \mathcal{M}[\![N_2]\!])$. No variables appear in subterms of $[N]_{CL}$ because N is closed. ∎

If we restrict our attention to programs, then the model $\mathcal{M}$ determines a *semantic definition* for L. The meaning of closed phrases easily generalizes to open phrases in the context of an environment $\mathcal{E}$ mapping each free variable x^τ to an element of $\mathcal{M}^\tau$.

Definition 2.5. (*Semantics of Open Phrases*) Let N be an open phrase in L and let $\mathcal{E}$ be an environment mapping each variable x^τ in $FV(N)$ to an element in $\mathcal{M}^\tau$. The *meaning* of N *determined by the model* $\mathcal{M}$ *and the environment* $\mathcal{E}$ (denoted $\mathcal{M}[\![N]\!]_\mathcal{E}$) is the algebraic meaning that $\mathcal{M}$ assigns to the combinatory term $[N]_{CL}$ in the environment $\mathcal{E}$. The *algebraic* meaning of a variable x^τ is simply $\mathcal{E}(x^\tau)$. ∎

Figure 1 gives the conventional *continuous function* model $\mathcal{C}$ for PCF; it interprets each type $\sigma \to \tau$ as the set of all *continuous* functions from $\mathcal{C}^\sigma$ into $\mathcal{C}^\tau$. We use this model to define the meaning of the language PCF.

Definition 2.6. (*Semantic Definition of PCF*) The semantic definition of PCF is the restriction of the meaning function $\mathcal{C}$ to programs. ∎

We will typically omit type superscripts from constants (as in Figure 1) where they can easily by reconstructed from context. In Figure 1 and the remainder of the paper, we rely on the following standard notation from domain theory.

Definition 2.7. (*Notation for Domains*) The symbol $\sqsubseteq$ denotes the approximation ordering on a domain; $\perp$ denotes the least element. The compound symbol $\mathbb{N}_\perp$ denotes the flat domain consisting of the natural numbers $\mathbb{N}$ augmented by $\perp$. For any domains $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \longrightarrow_c \mathcal{B}$ denotes the domain of *continuous* functions mapping $\mathcal{A}$ into $\mathcal{B}$. If f is a function mapping the domain D into itself, f^i denotes the i-fold composition of f with itself: $f^0 = \perp_D$ and $f^{i+1} = f \circ f^i$. In semantic definitions and proofs of theorems, the expression $\underline{\lambda}x \,.\, \cdots x \cdots$ denotes the function f defined by the equation $f(x) = \cdots x \cdots$. ∎

Domains:

$$\begin{array}{rcl} \mathcal{C}^o & = & \mathbb{N}_\perp \\ \mathcal{C}^{\sigma\to\tau} & = & [\mathcal{C}^\sigma \longrightarrow_c \mathcal{C}^\tau] \end{array}$$

Meaning Functions:

$$\begin{array}{rcl}
\mathcal{C}[\![e]\!] & = & \mathcal{C}[\![[e]_{CL}]\!] \\
\\
[\ulcorner n \urcorner]_{CL} & = & \ulcorner n \urcorner \\
[f]_{CL} & = & f \qquad (f \in \mathcal{F}) \\
[x]_{CL} & = & x \qquad (x \in V) \\
[(M_1\ M_2)]_{CL} & = & \mathsf{apply}([M_1]_{CL}, [M_2]_{CL}) \\
[\lambda x\,.\,M]_{CL} & = & \lambda^* x\,.\,[M]_{CL} \\
\\
\lambda^* x\,.\,x^\sigma & = & \mathsf{I} \\
\lambda^* x\,.\,M & = & \mathsf{apply}(\mathsf{K}, M) \qquad (x \notin FV(M)) \\
\lambda^* x\,.\,\mathsf{apply}(M_1, M_2) & = & \mathsf{apply}(\mathsf{apply}(\mathsf{S}, \lambda^* x\,.\,M_1), \lambda^* x\,.\,M_2) \\
\\
\mathcal{C}[\![\mathsf{apply}(f,a)]\!] & = & \mathcal{C}[\![f]\!](\mathcal{C}[\![a]\!]) \\
\mathcal{C}[\![\mathsf{I}]\!] & = & \underline{\lambda} x\,.\,x \\
\mathcal{C}[\![\mathsf{K}]\!] & = & \underline{\lambda} x\,.\,\underline{\lambda} y\,.\,x \\
\mathcal{C}[\![\mathsf{S}]\!] & = & \underline{\lambda} x\,.\,\underline{\lambda} y\,.\,\underline{\lambda} z\,.\,(x(z))(y(z)) \\
\mathcal{C}[\![\mathsf{Y}]\!] & = & \underline{\lambda} f\,.\,\bigsqcup\{f^i(\perp) \mid i \in \mathbb{N}\} \\
\mathcal{C}[\![\ulcorner n \urcorner]\!] & = & n \\
\\
\mathcal{C}[\![\mathsf{add1}]\!] & = & \underline{\lambda} m\,.\,\left\{\begin{array}{ll} m+1 & m \in \mathbb{N} \\ \perp & m = \perp \end{array}\right. \\
\\
\mathcal{C}[\![\mathsf{sub1}]\!] & = & \underline{\lambda} m\,.\,\left\{\begin{array}{ll} m-1 & m > 0 \\ \perp & m \in \{\perp, 0\} \end{array}\right. \\
\\
\mathcal{C}[\![\mathsf{if0}]\!] & = & \underline{\lambda} l\,.\,\underline{\lambda} m\,.\,\underline{\lambda} n\,.\,\left\{\begin{array}{ll} m & l = 0 \\ n & l > 0 \\ \perp & l = \perp \end{array}\right.
\end{array}$$

Figure 1: The Continuous Function Model $\mathcal{C}$ for PCF

2.2 Observational Equivalence, Sequentiality

We are finally ready to give a rigorous definition of the key concepts of denotational equivalence, observational equivalence, and full abstraction.

Definition 2.8. (*Denotational Equivalence, Observational Equivalence, Full Abstraction*) Two phrases, N and N', in the language L are *denotationally equivalent* in a model $\mathcal{M}$, written $N \equiv_{\mathcal{M}} N'$ iff N and N' have the same type and $\mathcal{M}[\![N]\!]_\mathcal{E} = \mathcal{M}[\![N']\!]_\mathcal{E}$ for all environments $\mathcal{E}$ mapping each variable x^τ in $FV(N) \cup FV(N')$ into $\mathcal{M}^\tau$. They are *observationally*

equivalent, written $N \simeq N'$, iff N and N' have the same type and for all contexts $C[\ ]$ such that $C[N]$ and $C[N']$ are programs, $\mathcal{M}[\![C[N]]\!] = \mathcal{M}[\![C[N']]\!]$. A model $\mathcal{M}$ is *fully abstract* iff $N \simeq N'$ implies $N \equiv_{\mathcal{M}} N'$ for closed N, N'.

The same definitions adapt in the obvious way to combinatory terms in L. ∎

Observational equivalence depends only on the meanings of programs. Hence, it is independent of the choice of model as long as models agree on the meanings of programs. Some references [18] in the literature define observational equivalence using a separate *operational* definition of the language L and prove that models for L are adequate. Our definition of observational equivalence, based on Milner's original definition [16], eliminates the need for two different forms of semantics and a proof of adequacy. Both definitions yield the same notion of observational equivalence.

The conventional model $\mathcal{C}$ for PCF is not fully abstract because the domain of continuous functions includes parallel functions like *parallel-and*, yet PCF can only define sequential functions. The procedures p_i defined in the introduction exploit this discrepancy: they behave differently only on inputs that are not sequential. Hence they have different denotations, but they are observationally equivalent.

Given a semantic definition $\mathcal{M}$ for a language L, we can define what it means for L to be *sequential*.

Definition 2.9. (*Sequentiality*) A language L with semantic definition $\mathcal{P}$ is *sequential* iff the following condition holds. Let C be a context and $M_1, \ldots, M_k$ be closed phrases such that $C[M_1, \ldots, M_k]$ is a program in L, $\mathcal{P}[\![C[M_1, \ldots, M_k]]\!] \in \mathbb{N}$, and $\mathcal{P}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{P}[\![\Omega]\!]$. Then there exists $j \in \mathbb{N}$, called a *sequentiality index* of $C[\ , \ldots, \ ]$, such that

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

for all expressions $M'_i, i \neq j$. ∎

This definition is based on Vuillemin's definition [26] of sequentiality for first order functions and Berry's generalization [2] of Vuillemin's definition to higher types. Plotkin proved that PCF is sequential [18:Activity Lemma] using an operational semantics for PCF that assigns the same meaning to PCF *programs* as the model $\mathcal{C}$.

3 Observing Sequentiality

PCF omits several language constructs that are essential in practical languages. One of the most glaring omissions is the lack of any provision for producing and handling run-time errors. Although this design choice simplifies the definition of the language, it has thwarted efforts to construct fully abstract models for PCF and other sequential languages. In the following paragraphs, we explain why the addition of two simple control mechanisms to PCF is important from the perspective of a language designer and how their presence in a programming language affects the construction of a denotational model.

3.1 Using errors, programmers can observe the order of evaluation.

The most obvious omission from PCF is a provision for computations to produce errors. A practical implementation of PCF would signal an error when the procedure sub1 is applied to $\ulcorner 0 \urcorner$ because $\ulcorner 0 \urcorner$ is an invalid input for sub1. Although this behavior is inconsistent with the semantics of PCF (which requires (sub1 $\ulcorner 0 \urcorner$) to diverge), it is far more informative from a programmer's point of view. More importantly, it permits programmers to observe the order of evaluation in programs by conducting simple experiments.

To provide PCF with error generators, we add two constants, error_1 and error_2, of ground type to the language and force all procedures in the extended language to be *error-sensitive*.[4] An operation is error-sensitive if it propagates any errors that it encounters in evaluating its arguments. In PCF with errors, a programmer can observe the evaluation order of subexpressions by exploiting the error-sensitivity of all program operations. For any pair of subexpressions, the programmer can replace each one by a distinct error value and observe the behavior of the modified program. If either subexpression is used in the evaluation of the original program, the modified program will return the error value corresponding to the subexpression that is evaluated first. As a result, the programmer can distinguish distinct versions of "commutative procedures" such as the addition procedures $+_l$ and $+_r$ defined in Section 2. The expression $(+_l\ \mathsf{error}_1\ \mathsf{error}_2)$ produces an error_1 because $+_l$ evaluates its left argument first; $(+_r\ \mathsf{error}_1\ \mathsf{error}_2)$ returns error_2 because $+_r$ evaluates its right argument first.

3.2 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 determine the order of evaluation among the arguments in a procedure application. To express these computations, the program must be able to delimit the dynamic extent of control actions such as errors [22]. For this reason, many practical languages include an error handling facility or a non-local control operator.

The original version of Scheme [24], for example, contained a lexically-scoped CATCH construct for implementing non-local exits from expressions. The evaluation of the expression

$$(\mathsf{CATCH}\ e\ (\mathsf{add1}\ (e\ 0)))$$

would return 0 by popping the control stack back through the lexical binding of e in (CATCH e ...) after encountering the sub-expression $(e\ 0)$. With such a CATCH construct, it is possible to write a program that determines the order of evaluation of a procedure's arguments, *e.g.*, the procedure

$$G = (\lambda f\,.(\mathsf{CATCH}\ x\ (f\ (x\ 0)\ (x\ 1))))$$

[4] We add two errors to PCF for pragmatic as well as technical reasons. Practical programming languages usually provide an **error** procedure that maps message strings to error elements containing the messages. Hence, there are many different error elements. From a technical perspective, we need two errors to make our model order-extensional. The model with one error is extensional, but not order-extensional.

maps $+_l$ to 0 and $+_r$ to 1.

To add the same capabilities to PCF, we introduce a family of procedures catch^τ of type $\tau \to o$. If f is a procedure of type $\tau = \tau_1 \to \ldots \tau_n \to o$, then $(\mathsf{catch}^\tau\ f)$ returns either $i-1$ if f evaluates the ith argument first or $k+n$ if t is a constant procedure with result k. Alternatively, $(\mathsf{catch}^\tau\ f)$ is $i-1$ if

$$(f\ \underbrace{\Omega\ \ldots\ \Omega}_{i-1}\ \mathsf{error}_j\ \underbrace{\Omega\ \ldots\ \Omega}_{n-i})$$

returns error_j for $j = 1, 2$. If $\tau = o$, catch^τ is degenerate: it always returns 0.

In PCF, the family of catch procedures has the same expressive power as Scheme's (downward) CATCH construct (see the corresponding note at the end of Section 4.4), but the catch procedures have a simpler definition that is better suited to proving the representability lemma (see Section 5).

We designate our sequential extension of PCF by the name *SPCF*.

Definition 3.1. (*SPCF*) *SPCF* is PCF augmented by the set of constants

$$\{\mathsf{error}_1, \mathsf{error}_2\} \cup \{\mathsf{catch}^\sigma \mid \sigma \in \mathit{Type}\}$$

where error_1 and error_2 have type o and catch^σ has type $(\sigma \to o)$. Henceforth, the symbol $\mathcal{F}$ will refer to the set

$$\{\ulcorner n \urcorner \mid n \in \mathbb{N}\} \cup \{\mathsf{add1}, \mathsf{sub1}, \mathsf{if0}, \mathsf{error}_1, \mathsf{error}_2\} \cup \{\mathsf{Y}^\sigma \mid \sigma \in \mathit{Type}\} \cup \{\mathsf{catch}^\sigma \mid \sigma \in \mathit{Type}\}$$

∎

Since the meaning of the procedure catch^σ depends only on the arity of the type σ, we will frequently drop the type subscript σ in favor of its arity k. The symbol catch_k denotes any procedure catch^σ where σ has the form $(\tau_1 \to \ldots \tau_k \to o)$.

3.3 Procedure denotations contain more computational structure than ordinary function graphs.

Conventional models for languages like SPCF are written in "continuation-passing style" to cope with the behavior of control operators. Since this form of model contains parallel functions, it is not fully abstract for sequential languages [22]. To construct a fully abstract model for SPCF, we want to develop a "direct" semantics that interprets procedures as functions.

The informal operational semantics of SPCF, notably the catch^τ operator, 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

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

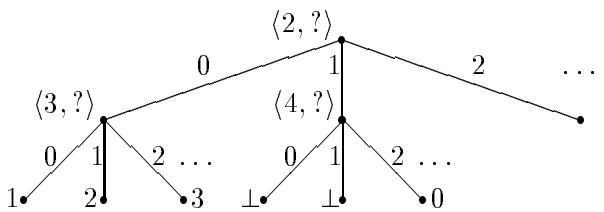
It has type $o \to (o \to (o \to (o \to o)))$, which means we can view it as a procedure mapping quadruples of integers to integers. The procedure p evaluates its second argument x first, but the subsequent evaluation order depends on the value of x. If x is 0, p evaluates

its third argument y next; otherwise it evaluates its fourth argument z. To capture this information, we can define the denotation of p 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 can be 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:

$$
\left\{\begin{array}{l}
\bot \mapsto \bot \\
\mathsf{error}_1 \mapsto \mathsf{error}_1 \\
\mathsf{error}_2 \mapsto \mathsf{error}_2 \\
\langle 2, \left\{\begin{array}{ll}
 & \left\{\begin{array}{l}\bot \mapsto \bot \\ \mathsf{error}_1 \mapsto \mathsf{error}_1 \\ \mathsf{error}_2 \mapsto \mathsf{error}_2 \\ 0 \mapsto 1 \\ 1 \mapsto 2 \\ \ldots\end{array}\right. \rangle \\
 & \left\{\begin{array}{l}\bot \mapsto \bot \\ \mathsf{error}_1 \mapsto \mathsf{error}_1 \\ \mathsf{error}_2 \mapsto \mathsf{error}_2 \\ 0 \mapsto \bot \\ 1 \mapsto \bot \\ 2 \mapsto 0 \\ 3 \mapsto 1 \\ \ldots\end{array}\right. \rangle
\end{array}\right. \\
\ldots
\end{array}\right.
$$

Since p is continuous and error-sensitive, it must map $\bot$ to $\bot$ and error_i to error_i.

If we rotate the preceding picture, it looks like a *decision tree*:



In the tree representation, we have added a question mark "?" after each argument index labeling an internal node. An internal node with the value $\langle i, ? \rangle$ represents the *initial* query about the ith argument. The rationale for this convention will become clear when we discuss procedures as arguments in the next paragraph. The edges below a node with value $\langle i, ? \rangle$ 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 $\mathbb{N} \cup \{\bot, \mathsf{error}_1, \mathsf{error}_2\}$), then p has completed its computation; otherwise, it continues to query its argument(s). In tree representations of procedures we omit branches that map $\bot$ to $\bot$ and error_i to error_i to avoid cluttering the representations with implied information.

3.4 Higher-order procedures explore trees sequentially.

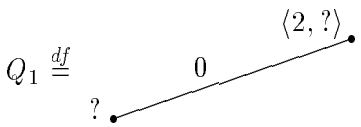
If we interpret procedures as decision trees, we must explain how higher-order procedures gather information about procedural inputs. Since trees resemble LISP S-expressions, we interpret higher-order procedures as processes that examine decision trees in essentially the same fashion as LISP procedures traverse S-expressions. To simplify notation, we express queries about procedural inputs as patterns that identify nodes in decision trees. More specifically, a query 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. The *response* produced by the query is the value of the specified node. If this match yields $\perp$ or error_i, the result of the entire process is $\perp$ or error_i. Like a LISP procedure, a higher-order procedure can inspect a node in an input tree only after inspecting its predecessors.

To illustrate this process, let us examine the behavior of the procedure

$$F = \lambda g\,.\,(\mathsf{add1}\,(g\ \Omega\ 0\ 2\ \mathsf{error}_1))\,.$$

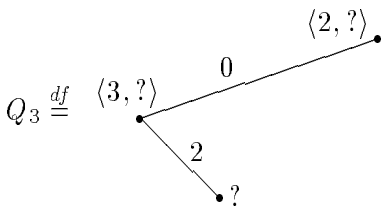
When (the tree representing) F is applied to an argument tree g, it knows nothing about the value of g. Since F must determine how g behaves on certain arguments, it asks the question $\langle 1, ?\rangle$ to determine the root of f. If the value of g is the procedure p described above, g answers this question with the response $\langle 2, ?\rangle$, which is the value of the root of p.

Now F knows that g initially demands information about its second argument. To find out more about g, 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 $\langle 1, Q_1\rangle$ where:



The query Q_1 is a pattern identifying the node value in g that F wants to inspect; it lies just below the root of g on the edge labeled 0. Matching the pattern Q_1 against p yields the node value $\langle 3, ?\rangle$, which is a query about p's third argument.

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



Matching this pattern against p yields the leaf value 3, which is a final answer from p. Since the subcomputation $(g\ \Omega\ 0\ 2\ \mathsf{error}_1)$ is now complete, F can add 1 to the result of calling g, yielding the final answer 4.

The preceding query-response protocol can be represented by a tree whose node values are queries and whose edges are labeled with the possible answers. Figure 2 shows part of the tree denoted by F, including the path that we just discussed. It also includes paths that

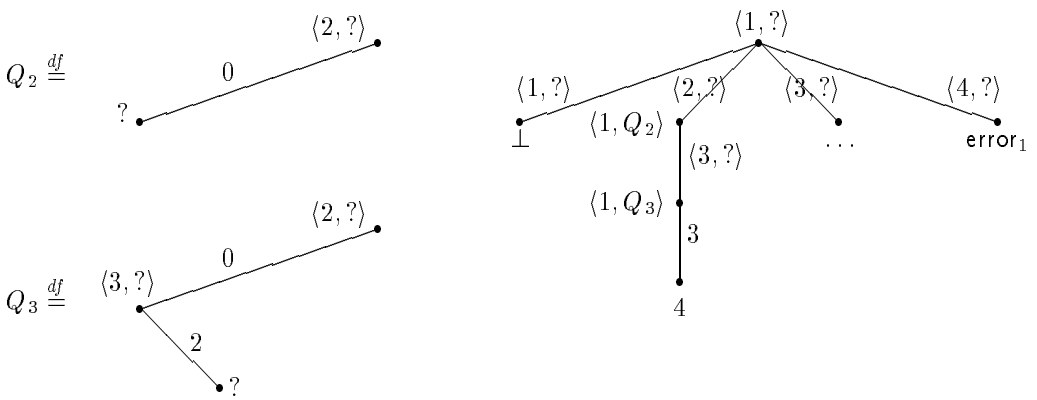
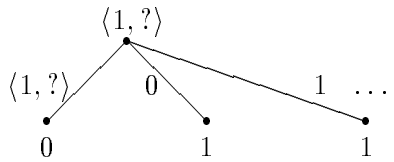


Figure 2: A Fragment of Procedure F

describe how F interacts with procedure arguments that immediately demand the value of their first or fourth argument: in the former case F returns $\bot$; in the latter case, F signals error_1. The other branches in the tree representing F have been omitted.

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:



Similarly, the catch procedure extracts information about the evaluation order of its argument. However, this information is not intensional because *distinct* trees represent *distinct* continuous functions. The presence of error values in the ground domain combined with the convention that all procedures are *error-sensitive* makes the decision tree representation *extensional*. In Section 4.2, we prove that there is a one-to-one correspondence between the graphs of observably sequential functions and their tree representations. Decision trees simply identify mathematical structure that is implicitly present in error-sensitive functions.[5]

[5]Berry and Curien [4] previously proposed representing procedures as trees in their work on *concrete sequential algorithms*. Since their domains do not include error elements, they associate distinct trees with the same function. For example, $+_l$ and $+_r$ correspond to distinct concrete sequential algorithms but they denote the same function [4:316]. In this framework, tree representations are *intensional*. In later work [5, 8], Berry and Curien noted that sequential algorithms can distinguish different algorithms for the same function, but they did not make a connection between intensional procedures and control operators like CATCH. Curien has recently shown that our model for SPCF is indeed closely related to the Berry-Curien model for PCF; the result will appear in a forthcoming paper [9].

4 The Tree Model

A model $\mathcal{M}$ of SPCF has essentially the same form as a model of PCF. $\mathcal{M}$ maps each type τ to a Scott domain $\mathcal{M}^\tau$, each constant $f \in \mathcal{F}$ of type τ to an element of $\mathcal{M}^\tau$, each auxiliary constant $o \in \mathcal{O}$ of type τ to an element of $\mathcal{M}^\tau$, and each function symbol $\mathsf{apply}^{\sigma,\tau} \in \mathcal{A}$ to a function $\mathcal{M}[\![\mathsf{apply}^{\sigma,\tau}]\!] : \mathcal{M}^{\sigma\to\tau} \times \mathcal{M}^\sigma \longrightarrow \mathcal{M}^\tau$. In our tree model of SPCF, called $\mathcal{T}$, each type τ is interpreted as a Scott domain of potentially infinite decision trees (where values of ground type are viewed as degenerate trees). Hence, the meaning of any closed SPCF phrase is a tree of the appropriate type. To define the meaning function $\mathcal{T}$, we extend the translation and abstraction algorithms, $[\cdot]_{CL}$ and λ^* for PCF (see Figure 1) in the obvious way. In contrast to the conventional model $\mathcal{C}$ for PCF, we interpret the constants in $\mathcal{O}$ as decision trees and the function symbols $\mathsf{apply}^{\sigma,\tau}$ as the functions $\mathcal{M}[\![\mathsf{apply}^{\sigma,\tau}]\!]$ on trees, which we will subsequently define. The functions $\mathsf{apply}^{\sigma,\tau}$ decode decision trees of type $\mathbb{T}^{\sigma\to\tau}$ into functions mapping $\mathbb{T}^\sigma$ into $\mathbb{T}^\tau$.

We can state this definition of the model $\mathcal{T}$ more formally as follows.

Definition 4.1. (*Tree model for SPCF*) $\mathcal{T}$ is the model for SPCF mapping:

1. each type τ to the tree domain $\mathbb{T}^\tau$ as defined in Section 4.1;
2. each constant c in the set

   $$\mathcal{O} \cup \mathcal{F} = \{\mathsf{S}^{\sigma,\tau,\upsilon}, \mathsf{K}^{\sigma,\tau}, \mathsf{I}^\tau\} \cup \{\mathsf{add1}, \mathsf{sub1}, \mathsf{if0}, \mathsf{error}_1, \mathsf{error}_2, \mathsf{catch}^\tau\} \cup \{\ulcorner n \urcorner \mid n \in \mathbb{N}\} \cup \{\mathsf{Y}^\sigma\}$$

   to the tree (in the appropriate domain) defined in Section 4.3; and
3. the function symbols $\mathsf{apply}^{\sigma,\tau}$ to the functions $\mathcal{T}[\![\mathsf{apply}^{\sigma,\tau}]\!]$ defined in Section 4.3.

∎

The remainder of this section of the paper focuses on filling in the gaps in this definition. The construction of the tree domains and the specific trees representing the combinators is quite intricate.

We will also show that the domain $\mathbb{T}^{\sigma\to\tau}$ of decision trees of type $\sigma \to \tau$ is isomorphic to the domain $\mathbb{F}^{\sigma\to\tau}$ of *error-sensitive* functions mapping $\mathbb{T}^\sigma$ into $\mathbb{T}^\tau$. Hence, the higher order domains in the model $\mathcal{T}$ can be interpreted as sets of functions.

In the following subsection, we define the tree domain $\mathbb{T}^\tau$ for each type τ. Next, we define the $\mathsf{apply}^{\sigma,\tau}$ function for each function type $\sigma \to \tau$ and prove that our tree representation is extensional: procedural trees are identical precisely when they cannot be distinguished by $\mathsf{apply}^{\sigma,\tau}$. In the third and fourth subsections, we assign interpretations to the primitive constants and the combinators of SPCF, and prove the basic equational properties of the tree model.

4.1 Tree Domains

A Scott domain is the ideal completion of a *finitary* basis [7, 20]. A finitary basis B is a countable, partially ordered set such that every finite, bounded subset has a least upper bound. The domain $\mathcal{D}_B$ determined by B is the set of ideals over B; the principal ideals of

B are the *finite elements* of the domain determined by B (the set of ideals over B). By this construction, a domain is a bounded-complete, ω-algebraic *cpo* (complete partial order). In discussing domains, it is convenient to ignore the distinction between the elements of the finitary basis and corresponding principal ideals.

Our definition of the domains for procedure types relies on the notion of *legal* queries and responses for arguments (which must have lower types than the procedure). Each tree domain $\mathbb{T}^\tau$ is the ideal completion of a finitary basis $\mathbb{D}^\tau$. In the process of defining the finitary bases $\mathbb{D}^\tau$, we will also define the set of legal queries $\mathbb{Q}^\tau$ of type τ (which higher-type objects can ask about elements of type τ) and the set of legal responses $\mathbb{R}^\tau$ to queries of type τ.

As a technical convenience, we will adopt a different format for responses in our definition of $\mathbb{D}^\tau$ than we used in the discussion in Section 3. Given the query q, an argument will return the response consisting of the path q with the "?" marker replaced by the value v of the matching node. In Section 3, an argument responded to a query q by simply returning the matching node value v instead of q with v substituted for "?". Both formats clearly contain the same new information, namely the value v. In the sequel, we will refer to the value v as the *answer* to the query and the modified query as the *response* to the query. In pictures of specific trees, we will still label edges with answers rather than responses to save space.

Since the precise definition of $\mathbb{D}^{\sigma\to\tau}$ is intricate, we will start by giving a preliminary definition that is intuitively clear but includes too many elements. Then we refine that definition to produce an extensional representation of functions as trees—a representation that establishes a one-to-one correspondence between decision trees and *error-sensitive* functions.

The preliminary definition of $\mathbb{D}^\tau$ for all types τ proceeds by induction on the structure of τ. In the following discussion, we will associate each higher order type $\tau_1 \to \ldots \to \tau_k \to o$ with the corresponding "uncurried" type $\tau_1 \times \ldots \times \tau_k \to o$ consisting of maps from appropriately typed k-tuples to ground values (natural numbers).

Base case: $\tau = o$: Expressions of type o denote either a natural number, the diverging computation ($\bot$), or an error value:

$$\mathbb{D}^o = \mathbb{N}_\bot^{\mathbb{E}} = \mathbb{N} \cup \mathbb{E}$$

where

$$\mathbb{E} = \{\bot, \mathsf{error}_1, \mathsf{error}_2\}.$$

The approximation ordering on $\mathbb{N}_\bot^{\mathbb{E}}$ is the usual one, that is, $\bot$ approximates all other elements, and the rest of the elements are mutually incomparable.

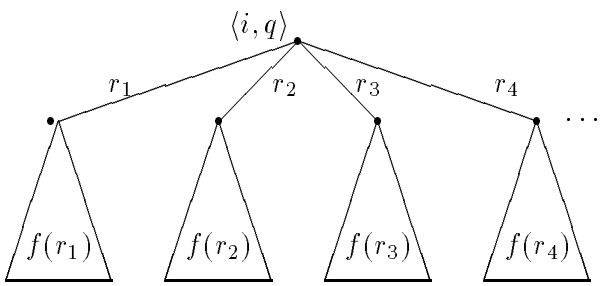
The only possible query about an argument of type o is the initial query "?" since the response completely determines the element. The only legal responses to this query are integers; the evaluation of an expression may produce an error element, but it cannot appear as a response in a tree. Consequently, we define the set of ground queries and ground responses as follows:

$$\mathbb{Q}^o = \{?\} \qquad \mathbb{R}^o = \mathbb{N}.$$

Inductive case: $\tau = \tau_1 \to \ldots \to \tau_k \to o$**:** Finite higher-order objects are simply finite decision trees. Every path within such a tree is finite and all leaves are elements of $\mathbb{N}_\perp^{\mathbb{E}}$. Similarly, only finitely many branches below each non-terminal node are not $\perp$. By hypothesis, we have already defined the sets $\mathbb{D}^{\tau'}$, $\mathbb{Q}^{\tau'}$, and $\mathbb{R}^{\tau'}$ for all types τ' that are structurally nested within the expression τ. We inductively define the preliminary form of the set $\mathbb{D}^\tau$ as follows. An element of type τ is either:

1. a leaf in $\mathbb{N}_\perp^{\mathbb{E}}$, or
2. a triple $\langle i, q, f\rangle$ consisting of an index i, $1 \le i \le k$, a query $q \in \mathbb{Q}^{\tau_i}$ (a query about the ith argument), and a *branching* function f that maps responses in $\mathbb{R}^{\tau_i}$ to simpler trees in $\mathbb{D}^\tau$. For all but a finite set of proper responses in $\mathbb{R}^{\tau_i}$, called the *proper domain* of f, the function f must produce the value $\perp$.

In pictorial form, we represent the tree $\langle i, q, f\rangle$ by the diagram:



In this picture, we have assumed that the proper domain of f has at least four elements $\{r_1, r_2, r_3, r_4\}$. The root of the subtree is a node with value $\langle i, q\rangle$. For each response r_j in the proper domain of f, there is an outgoing edge running from the root to the finite subtree representing $f(r_j)$. We sometimes refer to the *value* $\langle i, q\rangle$ at a node $\langle i, q, f\rangle$ as a *node*.

For each type τ, we give the following preliminary definitions for $\mathbb{Q}^\tau$ and $\mathbb{R}^\tau$. Queries and responses of type τ designate paths in trees of type τ. They have exactly the same form as finite trees with only one branch[6] with one important exception: all queries end with the special leaf "?". More precisely, the final branching function in a query must map the only response r in its proper domain to "?".

Within a path, we will represent the branching function that maps the response r to path p by the pair $\langle r, p\rangle$. In other words, $\langle r, p\rangle$ denotes the function f where $f(r) = p$ and $f(s) = \perp$ for $s \neq r$. Similarly, $\perp$ denotes the everywhere undefined branching function. A "?" at the end of each query identifies the requested node value. A response to query q replaces the the "?" marker in q with a final answer n in $\mathbb{N}$ or an intermediate answer $\langle i, q, \perp\rangle$. The notation $q[?/a]$ denotes the response to q obtained by replacing "?" with the answer a. An argument responds to the query q by returning $q[?/v]$ where v is the value of the node identified by q.

Figure 3 illustrates these conventions using the queries Q_2 and Q_3 presented in Section 3. R_2 and R_3 are legal responses to queries Q_2 and Q_3, respectively.

[6] In a tree with only one branch, the proper domain of each branching function f contains at most one element.

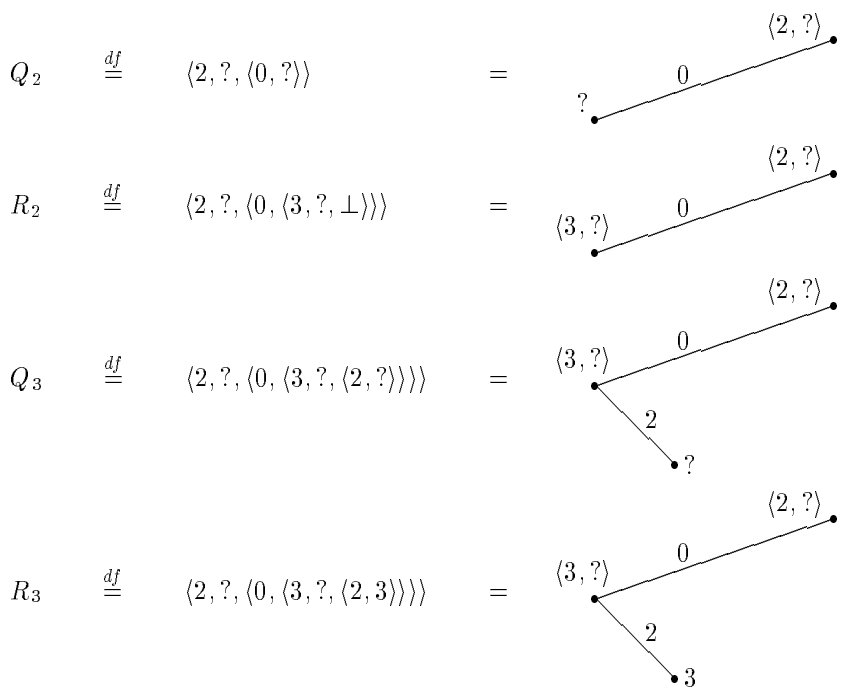


Figure 3: Representation of Queries and Responses

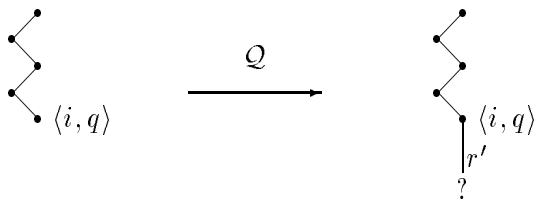
Every response r of type τ is a finite tree in $\mathbb{D}^\tau$. Every query q of type τ becomes an element of $\mathbb{D}^\tau$ if we replace "?" by $\bot$. For the sake of brevity, we will abbreviate the path $q[?/\bot]$ by the symbol $\overline{q}$. Similarly, in pictures of decision trees, we will usually label edges with answers instead of responses to save space.

Unfortunately, the preliminary definitions of $\mathbb{D}^\tau$, $\mathbb{Q}^\tau$, and $\mathbb{R}^\tau$ given above are too broad to constitute an extensional representation for error-sensitive functions. There are three sources for this imprecision. First, the definition of $\mathbb{D}^\tau$ permits duplicate queries in trees, implying that many different trees represent the same function. Second, the definition of trees does not require a procedure to inspect all the predecessors of a node in an argument tree before inspecting the node itself. Finally, the definition of $\mathbb{R}^\tau$ does not guarantee that the response to a query is consistent with previous responses.

We can correct the flaws in the preliminary definition of $\mathbb{D}^\tau$ by maintaining an environment that accumulates the responses that appear on a path from the root of a decision tree to a given node. We refer to this environment as the *tree context* of the node. We will frequently use the term *context* instead of tree context, when it causes no confusion. The set of contexts of type τ is denoted by the symbol $\mathbb{C}^\tau$. Technically, a context is a relation associating argument indices with responses; we will use it as a set-valued function from indices to sets of responses. A context contains the information that a procedure has gathered about its arguments at a given point in a computation. More precisely, the context ρ of a node α in a procedure P maps each argument index i to the the finite approximation $\bigsqcup \rho(i)$ in $\mathbb{D}^{\tau_i}$ that P has determined for argument i at node α.

Given the context ρ of a node α, we can determine the set of legal queries at α for each argument i as follows: a query q on argument i must extend the approximation tree $\bigsqcup \rho(i)$ for argument i by exactly one node, which is identified by a "?" in the query q. This restriction on queries guarantees that queries cannot be duplicated. It also guarantees that all the predecessors of a node in an argument tree are inspected before the node itself is inspected.

We will formalize the set of legal queries by defining a function $\mathcal{Q}$ that maps the set of prior responses for a particular index to the legal set of queries. For each previous legal response r, the function generates all queries that extend r by a legal response to the last query in r. Pictorially, we have:



for all legal responses r' to the query q. If the response r ends in a final answer rather than a query q, $\mathcal{Q}$ generates the empty set because there are no legal queries.

The last flaw in the preliminary definition of $\mathbb{D}^\tau$ that we need to correct is the definition of the set of legal responses to a query q. But this problem is easy to solve: a response $q[?/d]$ to q is legal iff the information d, either a node of the form $\langle j, p, \bot \rangle$ or an answer a, can appear at the point specified by q in the selected argument tree. This test reduces to confirming that the response $q[?/d]$ is a well-formed tree (path). Since arguments are trees of lower type, this reduction is inductive.

We are finally ready to define the finitary bases $\mathbb{D}^\tau$. For each type τ, we will define:

- a set of tree contexts $\mathbb{C}^\tau$,
- a function $\mathbb{D}^\tau$ mapping a tree context ρ to partial orders of finite *sub*trees that can appear in context ρ,
- a function $\mathbb{P}^\tau$ mapping a context ρ and a leaf set B to the set of paths over B starting in context ρ,
- a set $\mathbb{Q}^\tau$ of queries (in terms of $\mathbb{P}^\tau$),
- a set $\mathbb{R}^\tau$ of responses (in terms of $\mathbb{P}^\tau$),
- a function $\mathcal{Q}^\tau$ mapping a set of responses $R \subseteq \mathbb{R}^\tau$ to the set of *legal* queries extending R, and
- a function $\mathcal{R}^\tau$ mapping a query $q \in Q^\tau$ to the set of *legal* responses to q in $\mathbb{R}^\tau$.

In the process, we will define:

- a substitution function $\underline{\lambda} q, x \,.\, q[?/x]^\tau$ that replaces the ? at the end of $q \in \mathbb{Q}^\tau$ by the response $r \in \mathbb{R}^\tau$, and

- the partial function $\underline{\lambda} r, f \,.\, (r \,.\, f)^\tau$ that replaces the empty branching function $\perp$ at the end of a response $r \in \mathbb{R}^\tau$ by the branching function f.

Definition 4.2. ($\mathbb{D}^\tau$ *including* $\mathbb{C}^\tau$, $\mathbb{P}^\tau$, $\mathbb{Q}^\tau$, $\mathbb{R}^\tau$, $\mathcal{Q}^\tau$, $\mathcal{R}^\tau$, $\cdot[?/\cdot]^\tau$, *and* $(\cdot\,.\,\cdot)^\tau$) The definition proceeds by induction on the structure of τ.
Base Case: $\tau = o$. In the base case the definitions are degenerate:

- $\mathbb{C}^o = \emptyset$,
- $\mathbb{D}^o(\rho) = \mathbb{N}_\perp^{\mathsf{E}}$ (ρ must be $\emptyset$),
- $\mathbb{P}^o(\rho, B) = B$ (ρ must be $\emptyset$),
- $\mathbb{Q}^o = \{?\}$,
- $\mathbb{R}^o = \mathbb{N}$,
- $\mathcal{Q}^o(R) = \begin{cases} \emptyset & \text{if } R = \{?\} \\ \{?\} & \text{if } R = \emptyset \end{cases}$,
- $\mathcal{R}^o(q) = \mathbb{N}$ (q must be ?),
- $q[?/x]^o = x$ ($x \in \mathcal{R}^o(?)$), and
- $(r \,.\, f)^o$ is "undefined" for responses $r \in \mathbb{R}^o$ and branching functions f since no response in $\mathbb{R}^o$ ends in the empty branching function.

Induction Step: $\tau = \tau_1 \to \ldots \tau_k \to o$ where $k > 0$. By induction, we can assume that for all types σ (properly) nested within τ, the sets $\mathbb{C}^\sigma$, $\mathbb{Q}^\sigma$, $\mathbb{R}^\sigma$, the functions $\mathbb{D}^\sigma$, $\mathbb{P}^\sigma$, $\mathcal{Q}^\sigma$, $\mathcal{R}^\sigma$, and $\underline{\lambda} q, x \,.\, q[?/x]^\sigma$, and the partial function $\underline{\lambda} r, f \,.\, (r \,.\, f)^\sigma$ have already been defined. We define these sets, functions, and operations for type τ as follows.

Contexts: The set of *tree contexts* $\mathbb{C}^\tau$ is the least set of ordered pairs of the form $\{\langle i, r\rangle\}$ where $1 \le i \le k$ and $r \in \mathbb{R}^{\tau_i}$ satisfying the closure properties:

1. $\emptyset \in \mathbb{C}^\tau$;
2. $\rho \cup \{\langle i, r\rangle\} \in \mathbb{C}^\tau$ if $\rho \in \mathbb{C}^\tau$ and $r \in \mathcal{R}^{\tau_i}(q)$ for some $q \in \mathcal{Q}^{\tau_i}(\rho(i))$ where

$$\rho(i) \stackrel{df}{=} \{s \mid \langle i, s\rangle \in \rho\}\,.$$

Subtrees: For each context $\rho \in \mathbb{C}^\tau$, the partial order $\mathbb{D}^\tau(\rho)$ of *finite subtrees* is defined as follows. The universe of the $\mathbb{D}^\tau(\rho)$ is the least set satisfying the following closure properties:

1. $\mathbb{N}_\perp^{\mathsf{E}} \subseteq \mathbb{D}^\tau(\rho)$;
2. $\langle i, q, f\rangle \in \mathbb{D}^\tau(\rho)$ if $q \in \mathcal{Q}^{\tau_i}(\rho(i))$ and f is a partial function mapping responses in $\mathbb{R}^{\tau_i}$ to subtrees in $\mathbb{D}^\tau(\rho \cup \{\langle i, r\rangle\})$ such that the proper domain of f is a *finite* subset of $\mathcal{R}^{\tau_i}(q)$.

The definition of the universe of $\mathbb{D}^\tau(\rho)$ is succinctly summarized by the equation:

$$\begin{array}{rcl}\mathbb{D}^\tau(\rho) & = & \mathbb{N}_\perp^{\mathbb{E}} \cup \\ & & \{\langle i,q,f\rangle \mid \begin{array}{l} 1 \le i \le k,\ q \in \mathcal{Q}^{\tau_i}(\rho(i)),\ \{r \mid f(r) \neq \perp\} \text{ is finite,} \\ f : r \in \mathcal{R}^{\tau_i}(q) \mapsto d \in \mathbb{D}^\tau(\rho \cup \{\langle i,r\rangle\})\}\end{array}\end{array}$$

The approximation relation $\sqsubseteq$ for $\mathbb{D}^\tau(\rho)$ is the least relation satisfying the properties

- $\perp \sqsubseteq d$ for all $d \in \mathbb{D}^\tau(\rho)$.
- $\langle i,q,f\rangle \sqsubseteq \langle i,q,g\rangle$ if $f(r) \sqsubseteq g(r)$ for all $r \in \mathcal{R}^{\tau_i}(q)$.

It is easy to show that $\sqsubseteq$ is reflexive, antisymmetric and transitive. Hence $\mathbb{D}^\tau(\rho)$ is a partial order.

Finite Paths: For every tree context $\rho \in \mathbb{C}^\tau$ and every leaf set B, the set $\mathbb{P}^\tau(\rho,B)$ of *paths* is the least set satisfying the equation:

$$\begin{array}{rcl}\mathbb{P}^\tau(\rho,B) & = & B \cup \\ & & \{\langle i,q,\langle r,p\rangle\rangle \mid \begin{array}{l} 1 \le i \le k,\ q \in \mathcal{Q}^{\tau_i}(\rho(i)),\ r \in \mathcal{R}^{\tau_i}(q), \\ p \in \mathbb{P}^\tau(\rho \cup \{\langle i,r\rangle\},B)\}\end{array}\end{array}$$

The parameter B is the set of leaf values that can terminate paths. If $\perp \in B$, then the branching function $\langle r,\perp\rangle$ for any response r is synonymous with the empty branching function $\perp$. If $B \subseteq \mathbb{N}_\perp^{\mathbb{E}}$, then $\mathbb{P}^\tau(\rho,B) \subseteq \mathbb{D}^\tau(\rho)$.

Queries: The set $\mathbb{Q}^\tau$ of *queries* is the set of paths that end in the distinct marker "?":

$$\mathbb{Q}^\tau \quad = \quad \mathbb{P}^\tau(\emptyset,\{?\})\,.$$

Responses: The set $\mathbb{R}^\tau$ of *responses* is the set of non-empty paths ending in a final answer in $\mathbb{N}$ or the empty leaf function $\perp$:

$$\mathbb{R}^\tau \quad = \quad \mathbb{P}^\tau(\emptyset,\mathbb{N}_\perp) - \{\perp\}\,.$$

The definition of $\mathbb{R}^\tau$ exploits the fact that the branching function $\langle r,\perp\rangle$ is identical to the empty branching function $\perp$.

Auxiliary functions: The function $\underline{\lambda}q,x\,.\,q[?/x]^\tau$ is defined on queries $q \in \mathbb{Q}^\tau$ and responses $x \in \mathbb{R}^\tau$ as follows:

$$\begin{array}{rcl}?[?/x] & = & x \\ \langle i,q,\langle r,q'\rangle\rangle[?/x] & = & \langle i,q,\langle r,q'[?/x]\rangle\rangle\end{array}$$

In other words, the expression $q[?/x]$ denotes the path consisting of the query q with the terminating "?" replaced by the tree x. We use the abbreviation $\overline{q}$ for $q[?/\perp]$.

The partial function $\underline{\lambda}q,x\,.\,(r\,.\,f)^\tau$ is defined for responses $r \in \mathbb{R}^\tau$ that end with the empty branching function $\perp$:

$$\begin{array}{rcl}\langle i,q,\perp\rangle\,.\,f & = & \langle i,q,f\rangle \\ \langle i,q,\langle r,r'\rangle\rangle\,.\,f & = & \langle i,q,\langle r,r'\,.\,f\rangle\rangle\,.\end{array}$$

Legal Queries: For any set of responses $R \subseteq \mathbb{R}^\tau$, the set $\mathcal{Q}^\tau(R)$ of *legal queries* is the subset of $\mathbb{Q}^\tau$ defined the equation

$$\mathcal{Q}^\tau(R) = \begin{cases} \{?\} & \text{if } R = \emptyset \\ \{q \in \mathbb{Q}^\tau \mid \neg\exists r\, [q \sqsubset r \sqsubseteq (\bigsqcup R)],\ \exists r \in R\, (q = r\,.\langle r', ?\rangle)\} & \text{if } R \neq \emptyset \end{cases} .$$

Legal Responses: For any query $q \in \mathbb{Q}^\tau$, the set $\mathcal{R}^\tau(q)$ of *legal responses* is the subset of $\mathbb{R}^\tau$ defined by the equation

$$\mathcal{R}^\tau(q) = \{r \in \mathbb{R}^\tau \mid r = q[?/a] \text{ for } a \in \mathbb{N} \text{ or } r = q[?/\langle i, p, \bot\rangle]\} .$$

The *partial order* $\mathbb{D}^\tau$ of finite trees of type τ is defined as $\mathbb{D}^\tau(\emptyset)$. ∎

It is easy to show that the set $\mathbb{D}^\tau$ forms a finitary basis under the approximation ordering $\sqsubseteq$.

Lemma 4.3 *$\mathbb{D}^\tau$ is a finitary basis.*

Proof. $\mathbb{D}^\tau$ is a partial order because $\sqsubseteq$ is reflexive, anti-symmetric, and transitive. $\mathbb{D}^\tau$ is countable because every branching function has a finite proper domain. It is straightforward but tedious to prove that every finite bounded subset of $\mathbb{D}^\tau$ has a least upper bound. ∎

As an immediate consequence of this lemma, we conclude that the ideal-completion of $\mathbb{D}^\tau$ is a Scott domain (ω-algebraic bounded-complete *cpo*). The completion process adds *infinite* trees as the limits of ascending chains of finite elements in each domain $\mathbb{D}^\tau$. The result is the Scott domain designated $\mathbb{T}^\tau$. Recall that a directed set is non-empty and contains an upper bound for every pair of elements.

Theorem 4.4 *Let $\mathcal{I}_e$ denote the principal ideal $\{d \in \mathbb{D}^\tau \mid d \sqsubseteq e\}$. For each type τ, the set $\mathbb{T}^\tau$ of ideals over $\mathbb{D}^\tau$ forms a bounded-complete, ω-algebraic cpo under set inclusion $\subseteq$:*

1. *Every directed subset of $\mathbb{T}^\tau$ has a least upper bound in $\mathbb{T}^\tau$; every bounded subset of $\mathbb{T}^\tau$ has a least upper bound.*
2. *The finite elements of $\mathbb{T}^\tau$ are precisely the principal ideals over $\mathbb{D}^\tau$: $\{\mathcal{I}_d \mid d \in \mathbb{D}^\tau\}$.*
3. *For every ideal $\mathcal{I} \in \mathbb{T}^\tau$, the set $\{\mathcal{I}_d \mid d \in \mathbb{D}^\tau, \mathcal{I}_d \subseteq \mathcal{I}\}$ is directed and*

$$\mathcal{I} = \bigsqcup \{\mathcal{I}_d \mid d \in \mathbb{D}^\tau, \mathcal{I}_d \subseteq \mathcal{I}\}.$$

Proof. The three properties are a consequence of Lemma 4.3 and a general theorem for ideal completions of finitary bases [7]. ∎

As the last topic in our discussion of the domain theory underlying our model, we will prove a lemma asserting that for every pair f and g of distinct trees in $\mathbb{D}^\tau$, there is a path p such that the subtrees in f and g identified by p have distinct roots. We will use this fact in the proof of several lemmas and theorems later in the paper.

Before we state and prove the lemma, we need to introduce some new terminology and notation. First we need to define the subtree $d@p$ at position $p \in \mathbb{Q}^\tau$ in the tree $d \in \mathbb{D}^\tau$. In the process, we will also define some important relations on queries and trees.

Definition 4.5. (@ *operator, valid query, query prefix*) Given a query $q \in \mathbb{Q}^\tau$ and a tree $d \in \mathbb{D}^\tau$, $d@q$ denotes the subtree identified by the "?" marker in q. More formally, @ is the least *partial* function mapping $\mathbb{Q}^\tau \times \mathbb{D}^\tau$ into $\bigcup_{\rho \in \mathbb{C}^\tau} \mathbb{D}^\tau(\rho)$ that satisfies the equations:

$$\begin{aligned} d@? &= d \\ \langle i, p, f\rangle @ \langle i, p, \langle r, q'\rangle\rangle &= f(r)@q' . \end{aligned}$$

If $d@q$ is defined for $d \in \mathbb{D}^\tau$ and $q \in \mathbb{Q}^\tau$, then we say q is a *valid query* in d; the condition holds precisely when $\overline{q} \sqsubseteq d$. If $\overline{p} \sqsubset q$ for $p \in \mathbb{Q}^\tau$, we say that p is a *prefix* of q. If $\overline{q} \sqsubseteq d$ and $\overline{p} \not\sqsubseteq d$ for all $p \sqsupset q$, then $d@q = \bot$ and $d@p$ is undefined for all $p \sqsupset q$. Note that $d@q = \bot$ does not mean that $d@q$ is undefined; $\bot$ is a legitimate value in the range of the partial function @. When $d@q = \bot$, we say that q *probes the perimeter* of d. ∎

Next we need to define what it means for two subtrees f and g to have different roots. We say that f and g are *immediately incomparable* iff they have different roots.

Definition 4.6. (root *operator, immediately incomparable subtrees*) Given a subtree $d \in \mathbb{D}^\tau(\rho)$ for some tree context ρ, the root $\mathsf{root}(d)$ of d is defined as follows:

$$\begin{aligned} \mathsf{root}(a) &= a \qquad a \in \mathbb{N}_\bot \\ \mathsf{root}(\langle i, q, f\rangle) &= \langle i, q\rangle \quad \langle i, q, f\rangle \in \mathbb{D}^\tau(\rho) . \end{aligned}$$

Two subtrees $d, e \in \mathbb{D}^\tau(\rho)$ are *immediately incomparable* iff $\mathsf{root}(d) \neq \mathsf{root}(e)$. They are *immediately comparable* iff $\mathsf{root}(d) = \mathsf{root}(e)$. ∎

Immediately incomparable trees are obviously incomparable under the approximation relation $\sqsubseteq$. However, immediately comparable trees may not be comparable under $\sqsubseteq$.

Given this terminology, we can formally state the lemma asserting that distinct trees have immediately incomparable subtrees.

Lemma 4.7 *For $f, g \in \mathbb{D}^\tau$, $f \not\sqsubseteq g$ implies that there is a query p such that $f@p$ and $g@p$ are both defined, $f@p \not\sqsubseteq g@p$, and the subtrees $f@p$ and $g@p$ are immediately incomparable.*

Proof. The proof of the lemma hinges on the following claim.

Claim 4.8 *For $d \in \mathbb{D}^\tau$, $d = \bigsqcup \{q[?/a] \mid q \in \mathbb{Q}^\tau,\ q[?/a] \sqsubseteq d,\ a \in \mathbb{N}_\bot^{\mathsf{E}}\}$.*

Proof (of Claim). We prove that the claim holds for all tree domains $\mathbb{D}^\tau(\rho)$. The proof proceeds by induction on the size (structure) of d. Let $d \in \mathbb{D}^\tau(\rho)$ for some context ρ. If $d \in \mathbb{N}_\bot^{\mathsf{E}}$, then there is only one q such that $\overline{q} \sqsubseteq d$, namely $q = ?$. In this case, the lemma obviously holds. If $d \notin \mathbb{N}_\bot^{\mathsf{E}}$, then d must have the form $\langle i, p, f\rangle$ where i is an argument index, p is a query in $\mathbb{Q}^{\tau_i}$ and f is a branching function. For each response r in the proper domain of f, there is a subtree $d_r = f(r)$ in $\mathbb{D}^\tau(\rho \cup \{\langle i, r\rangle\})$ for which the inductive hypothesis holds, *i.e.*,

$$d_r = \bigsqcup \{q[?/(d_r@q)] \mid q \in \mathbb{Q}^\tau,\ \overline{q} \sqsubseteq d_r,\ (d_r@q) \in \mathbb{N}_\bot^{\mathsf{E}}\} .$$

If we insert each of the subpaths q of d_r in the hole in the tree prefix $\langle i, p, [\ ]\rangle$, we obtain the required set of paths for d:

$$\begin{aligned}
d &= \langle i, p, \{\langle r, d_r\rangle \mid r \in \mathcal{R}^{\tau_i}(p)\}\rangle \\
&= \langle i, p, \{\langle r, \bigsqcup\{q[?/(d_r@q)] \mid q \in \mathbb{Q}^\tau, \overline{q} \sqsubseteq d_r, (d_r@q) \in \mathbb{N}_\perp^{\mathbb{E}}\}\rangle \mid r \in \mathcal{R}^{\tau_i}(p)\}\rangle \\
&= \bigsqcup\{\langle i, p, \langle r, q[?/(d@q)]\rangle\rangle \mid q \in \mathbb{Q}^\tau, \overline{q} \sqsubseteq d_r, (d_r@q) \in \mathbb{N}_\perp^{\mathbb{E}}, r \in \mathcal{R}^{\tau_i}(p)\} \\
&= \bigsqcup\{q[?/(d@q)] \mid q \in \mathbb{Q}^\tau, \overline{q} \sqsubseteq d, (d_r@q) \in \mathbb{N}_\perp^{\mathbb{E}}\} \\
&= \bigsqcup\{q[?/a] \mid q \in \mathbb{Q}^\tau, q[?/a] \sqsubseteq d, a \in \mathbb{N}_\perp^{\mathbb{E}}\}\ .
\end{aligned}$$

Since $\mathbb{D}^\tau = \mathbb{D}^\tau(\emptyset)$, the claim is proved. ∎ (Claim)

By this claim, we know there is a query $q \in \mathbb{Q}^\tau$ and a leaf $a \in \mathbb{N}_\perp^{\mathbb{E}}$ such that $q[?/a] \sqsubseteq f$ but $q[?/a] \not\sqsubseteq g$. Otherwise, $f \sqsubset g$. Let p be the longest prefix of q such that $f_p = f@p$ and $g_p = g@p$ are both defined. Such a prefix must exist because $f@?$ and $g@?$ are both defined and $f@? \neq \perp$. Clearly, $f_p \not\sqsubseteq g_p$, since $q[?/a] \not\sqsubseteq g$. Assume that f_p and g_p are immediately comparable. If $f_p, g_p \in \mathbb{N}_\perp^{\mathbb{E}}$, then $f_p = g_p$ and $p = q$, contradicting the fact that $f_p \not\sqsubseteq g_p$. If $f_p = \langle i, q, f'\rangle$ and $g_p = \langle i, q, g'\rangle$, then $f' \not\sqsubseteq g'$ since $f_p \not\sqsubseteq g_p$. Let p' be the query $p[?/\langle i, q, \langle r, ?\rangle\rangle]$ where r is chosen so that p' a prefix of q. The query p' is a longer prefix of q than p, yet p' is valid in both f and g, which contradicts the definition of p. Hence, f_p and g_p must be immediately incomparable. ∎

4.2 Application

A critical component of any model of a language L based on the simply typed λ-calculus is the interpretation of the function symbols $\mathsf{apply}^{\sigma,\tau}$. In the tree model $\mathcal{T}$, the function $\mathcal{T}[\![\mathsf{apply}^{\sigma,\tau}]\!] : \mathbb{T}^{\sigma\to\tau} \times \mathbb{T}^\sigma \longrightarrow \mathbb{T}^\tau$ decodes decision trees into functions. For the sake of brevity we will drop the type superscript in subsequent references to $\mathsf{apply}^{\sigma,\tau}$. Similarly, we write apply instead of $\mathcal{T}[\![\mathsf{apply}]\!]$ where the usage is clear from context.

For inputs f in $\mathbb{T}^{\sigma\to\tau}$ and x in $\mathbb{T}^\sigma$, $\mathsf{apply}(f, x)$ is defined by case analysis on f:

1. If f is a constant, apply ignores the value of x and returns the value f.

2. If f has the form $\langle 1, q, g\rangle$, apply matches q against the argument x. If x returns a proper response r, apply selects the appropriate branch in the branching function, $g(r)$, and continues the process of constructing the output tree. If x returns an improper answer in $\mathbb{E} = \{\perp, \mathsf{error}_1, \mathsf{error}_2\}$, apply returns that answer.

3. If f has the form $\langle i+1, q, g\rangle$ for $i \geq 1$, apply must produce the tree representing the function $f(x)$; it has the form $\langle i, q, \underline{\lambda} r\,.\,\mathsf{apply}(g(r), x)\rangle$. But $g(r)$ is a subtree rather than a tree, indicating that we must generalize apply to subtrees.

To define apply formally, we must generalize the informal definition given above to subtrees. In addition, since the domain and co-domain of apply include limit points (infinite trees), we will define apply as the continuous extension of an auxiliary function apply'.

Definition 4.9. (apply) The function $\mathsf{apply} : \mathbb{T}^{\sigma\to\tau} \times \mathbb{T}^\sigma \longrightarrow \mathbb{T}^\tau$ is constructed from the auxiliary partial function apply' as follows. For any tree context ρ of arity k, let $\rho'(i) =$

$\rho(i+1), 1 \leq i < k$. The auxiliary partial function apply' maps $\mathbb{D}^{\sigma\to\tau}(\rho) \times \mathbb{D}^{\sigma}$ into $\mathbb{D}^{\tau}(\rho')$. For all $f \in \mathbb{T}^{\sigma\to\tau}$ and $d \in \mathbb{T}^{\sigma}$,

$$\mathsf{apply}(f,d) = \bigsqcup \{\mathsf{apply}'(f',d') \mid f' \sqsubseteq f, f' \in \mathbb{D}^{\sigma\to\tau}(\emptyset), d' \sqsubseteq d, d' \in \mathbb{D}^{\sigma\to\tau}\}\,.$$

The function $\mathsf{apply}' : \mathbb{D}^{\sigma\to\tau}(\rho) \times \mathbb{D}^{\sigma} \longrightarrow \mathbb{D}^{\tau}(\rho')$ is defined by the following recursion equations:

$$\begin{aligned}
\mathsf{apply}'(f,d) &= f \qquad \text{if } f \in \mathbb{N}_{\perp}^{\mathbb{E}} \\
\mathsf{apply}'(\langle 1,q,g\rangle,d) &= \begin{cases} d@q & \text{if } d@q \in \mathbb{E} \\ \mathsf{apply}'(g(q[?/d@q]),d) & \text{if } d@q \in \mathbb{N} \\ \mathsf{apply}'(g(q[?/\langle j,p\rangle]),d) & \text{if } d@q = \langle j,p,d_1'\rangle \end{cases} \\
\mathsf{apply}'(\langle i+1,q,g\rangle,d) &= \langle i,q,\underline{\lambda} r_i\,.\,\mathsf{apply}'(g(r_i),d)\rangle
\end{aligned}$$

∎

It is easy to show that apply' produces a well-defined output in $\mathbb{D}^{\tau}(\rho')$ for $f \in \mathbb{D}^{\sigma\to\tau}(\rho)$ and $d \in \mathbb{D}^{\sigma}$ such that $\bigsqcup \rho(1) \sqsubseteq d$. The behavior of apply' on finite trees $d \in \mathbb{D}^{\sigma}$ where $\bigsqcup \rho(1) \not\sqsubseteq d$ is irrelevant because apply depends only on apply' where $\rho = \emptyset$. Assume $\sigma = \tau_1$ and $\tau = \tau_2 \to \ldots \to \tau_k \to o$. The proof that apply' is well-defined for d such that $\bigsqcup \rho(1) \sqsubseteq d$ proceeds by induction on the size of the finite element f. The induction hypothesis consists of two parts:

1. For all $f \in \mathbb{D}^{\tau_1\to\ldots\to\tau_k\to o}(\rho)$ and $d \in \mathbb{D}^{\tau_1}$ such that $\bigsqcup \rho(1) \sqsubseteq d$,

   $$\mathsf{apply}'(f,d) \in \mathbb{D}^{\tau_2\to\ldots\to\tau_k\to o}(\rho')$$

   where $\rho'(i) = \rho(i+1)$ for $1 \leq i < k$.

2. If $f = \langle 1,q,g\rangle$, then $\overline{q} \sqsubseteq d$, implying that $d@q$ is always well-defined in the definition of apply' when $\bigsqcup \rho(1) \sqsubseteq d$.

In addition, it is straightforward to check that for all $f \in \mathbb{T}^{\sigma\to\tau}$ and $d \in \mathbb{T}^{\sigma}$, the set $\{\mathsf{apply}'(f',d') \mid f' \sqsubseteq f, f' \in \mathbb{D}^{\sigma\to\tau}, d' \sqsubseteq d, d' \in \mathbb{D}^{\sigma\to\tau}\}$ is directed, implying that it has a least upper bound. Hence, apply is a well-defined, continuous function from pairs of trees to trees. To simplify notation, we will identify the functions apply and apply' in the rest of the paper. Note that apply' is defined for finite subtrees as well as finite trees.

A model for a language based on the typed λ-calculus is extensional when procedure denotations are identical if and only if they behave identically under application. The model is order-extensional if this property also holds for approximations between procedure denotations.

Definition 4.10. (*Extensionality, Order-extensionality*) Let L be a language conforming to the syntax of the typed λ-calculus with constants. A *model* $\mathcal{M}$ for L is *extensional* iff for all $f, g \in \mathcal{M}^{\sigma\to\tau}$, $\mathsf{apply}(f,d) = \mathsf{apply}(g,d)$ for all $d \in \mathcal{M}^{\sigma}$ implies $f = g$. It is *order-extensional* iff for all $f, g \in \mathcal{M}^{\sigma\to\tau}$, $\mathsf{apply}(f,d) \sqsubseteq \mathsf{apply}(g,d)$ for all $d \in \mathcal{M}^{\sigma}$ implies $f \sqsubseteq g$.

∎

Although the interpretations in our model $\mathcal{T}$ for the constants and combinators of SPCF are still unspecified we can prove that the model is extensional and order-extensional,[7] because these properties depend only on the definition of the domains $\mathbb{T}^\tau$ and the functions $\mathsf{apply}^{\sigma,\tau}$.

Theorem 4.11 *$\mathcal{T}$ is extensional and order-extensional.*

Proof. Since extensionality is obviously implied by order-extensionality, we will only prove the second claim. Assume $f \not\sqsubseteq g$ for $f, g \in \mathbb{T}^{\sigma\to\tau}$. We will show that there exists $d \in \mathbb{T}^\sigma$ such that

$$\mathsf{apply}(f,d) \not\sqsubseteq \mathsf{apply}(g,d).$$

Since the domain $\mathbb{T}^{\sigma\to\tau}$ is algebraic,

$$f = \bigsqcup\{f' \mid f' \sqsubseteq f, f' \in \mathbb{D}^{\sigma\to\tau}\} \text{ and } g = \bigsqcup\{g' \mid g' \sqsubseteq g, g' \in \mathbb{D}^{\sigma\to\tau}\}\,.$$

Consequently there exists a finite $f' \sqsubseteq f$ such that $f' \not\sqsubseteq g$. Moreover, for all finite $g' \sqsubseteq g$, $f' \not\sqsubseteq g'$. By Lemma 4.16 below, there exists a finite $d \in \mathbb{D}^\tau$ such that

$$\mathsf{apply}(f',d) \not\sqsubseteq \mathsf{apply}(g',d)$$

for all $g' \sqsubseteq g$. From the definition of apply, we know that $\mathsf{apply}(f',d)$ is finite and continuous. Hence,

$$\mathsf{apply}(f',d) \not\sqsubseteq \bigsqcup\{\mathsf{apply}(g',d) \mid g' \sqsubseteq g'\} = \mathsf{apply}(g,d)\,.$$

By monotonicity,

$$\mathsf{apply}(f',d) \sqsubseteq \mathsf{apply}(f,d)\,,$$

implying

$$\mathsf{apply}(f,d) \not\sqsubseteq \mathsf{apply}(g,d)\,.$$

which is precisely what we needed to prove. ∎

To complete the proof of the theorem, we need to state and prove a lemma asserting that extensionality holds for finite decision trees. The proof relies on the following observation. If f and g are distinct finite trees, then there must be a path from the root to a node position where the trees have different node values, *e.g.*, one tree probes its ith argument, while the other is a constant function. If we apply f and g to an argument d corresponding to this path, we can show that f and g produce inconsistent results on the input d.

To simplify the proof of Lemma 4.14 and the proof of the full abstraction theorem in the next section, we introduce some new terminology and prove the extensionality lemma with the help of an auxiliary lemma. First, we define the tree context determined by a path.

[7] P.L. Curien (ENS) [personal communication, August 1991] pointed out that the order-extensionality theorem does not hold if the model only contains one error value. For example, the functions error and $\langle 1, ?, \underline{\lambda}x\,.\,\mathsf{error}\rangle$ are incomparable, yet for all d, $\mathsf{apply}(\langle 1, ?, \underline{\lambda}x\,.\,\mathsf{error}\rangle, d) \sqsubseteq \mathsf{apply}(\mathsf{error}, d)$. On the other hand, while a model based on a single error value is not order-extensional, it is still extensional. It is also fully abstract.

Definition 4.12. (*Tree context determined by a query*) Let $\tau = \tau_1 \to \ldots \tau_k \to o$. A query $q \in \mathbb{Q}^\tau$ determines the *tree context* $\widehat{q} = R(q, \emptyset)$ where $R : \mathbb{Q}^\tau \times \mathbb{C}^\tau \longrightarrow \mathbb{C}^\tau$ is the least function satisfying the equations:

$$\begin{array}{rcl} R(?, \rho) & = & \rho \\ R(\langle i, p, \langle r, q' \rangle\rangle, \rho) & = & R(q', \rho \cup \{\langle i, r \rangle\}) \end{array}$$

∎

Second, we define a function $shift_1$ that maps paths of arity k to the corresponding paths of arity $k-1$. The function $shift_1$ simply eliminates queries about the first argument and converts queries about argument queries about argument $i+1$ into queries about argument i.

Definition 4.13. ($shift_1$) The function $shift_1$ from queries of type

$$\tau_1 \to \ldots \to \tau_k \to o$$

to queries of type

$$\tau_2 \to \ldots \to \tau_k \to o$$

is defined as follows:

$$\begin{array}{rcl} shift_1(\langle 1, p, \langle r, q \rangle\rangle) & = & shift_1(q) \\ shift_1(\langle i+1, p, \langle r, q \rangle\rangle) & = & \langle i, p, \langle r, shift_1(q) \rangle\rangle \\ shift_1(?) & = & ? \end{array}$$

The function also has a natural extension to paths that end in final answers:

$$shift_1(a) = a \quad a \in \mathbb{N}.$$

∎

Based on the two preceding definitions, it is now possible to state the lemma that relates applications of elements and paths to the context information determined by the path.

Lemma 4.14 *Let $d \in \mathbb{D}^\tau$, let $q \in \mathbb{Q}^\tau$ be a query that determines the context $\widehat{q}$, and let e be a subtree in $\mathbb{D}^\tau(\widehat{q})$. If $d@q = e$ and $d_1 \sqsupseteq \bigsqcup \widehat{q}(1)$, then*

$$\mathsf{apply}(d, d_1)@shift_1(q) = \mathsf{apply}(e, d_1)\,.$$

Proof. The proof proceeds by induction on the query q. In the base case, $q = ?$, implying $d = e$ $shift_1(q) = ?$, so the lemma obviously holds. In the induction step, q has the form $\langle i, p, \langle r, q' \rangle\rangle$ implying that $d = \langle i, p, f \rangle$ where $f(r)@q' = e$ and $d_1 \sqsupseteq \bigsqcup \widehat{q'}(1)$. If $i = 1$, then $shift_1(q) = shift_1(q')$. In addition, $d_1 \sqsupseteq \bigsqcup \widehat{q}(1)$ implies that $r \sqsubseteq d_1$. As a result, we can transform $\mathsf{apply}(d, d_1)@shift_1(q)$ as follows:

$$\begin{array}{rcl} \mathsf{apply}(d, d_1)@shift_1(q) & = & \mathsf{apply}(\langle i, p, f \rangle, d_1)@shift_1(q) \\ & = & \mathsf{apply}(f(r), d_1)@shift_1(q') \\ & & \qquad \text{(by definition of } \mathsf{apply} \text{ and } shift_1) \\ & = & \mathsf{apply}(e, d_1) \qquad \text{(by induction)}\,, \end{array}$$

proving the lemma holds when $i = 1$. If $i > 1$, then $\mathit{shift}_1(q) = \langle i-1, p, \langle r, \mathit{shift}_1(q')\rangle\rangle$. As a result, we can transform $\mathsf{apply}(d, d_1)@\mathit{shift}_1(q)$ as follows:

$$\begin{aligned}
\mathsf{apply}(d, d_1)@\mathit{shift}_1(q) &= \mathsf{apply}(\langle i, p, f\rangle, d_1)@\mathit{shift}_1(q) \\
&= \langle i-1, p, \underline{\lambda} s\,.\,\mathsf{apply}(f(s), d_1)\rangle@\langle i-1, p, \langle r, \mathit{shift}_1(q')\rangle\rangle \\
&\qquad \text{(by definition of apply and } \mathit{shift}_1\text{)} \\
&= \mathsf{apply}(f(r), d_1)@\mathit{shift}_1(q') \qquad \text{(by definition of @)} \\
&= \mathsf{apply}(e, d_1) \qquad \text{(by induction)},
\end{aligned}$$

completing the proof of the lemma. ∎

The lemma implies a simple corollary about the application of paths to all possible arguments.

Corollary 4.15 *Let $\tau = \tau_1 \to \ldots \tau_k \to o$. Let $q \in \mathbb{Q}^\tau$ be a query that determines the context $\widehat{q}$, and let e be a subtree in $\mathbb{D}^\tau(\widehat{q})$. If $d \sqsupseteq q[?/e]$ and $d_i \sqsupseteq \bigsqcup \widehat{q}(i)$ for i, $1 \le i \le k$,*

$$\mathsf{apply}(d, d_1, \ldots, d_k) = \mathsf{apply}(q[?/e], d_1, \ldots, d_k) = \mathsf{apply}(e, d_1, \ldots, d_k).$$

We are finally ready to prove that the model $\mathcal{T}$ is order-extensional and hence extensional.

Lemma 4.16 *Let f, g be elements in $\mathbb{D}^{\sigma\to\tau}$. If for all finite $d \in \mathbb{D}^\sigma$, $\mathsf{apply}(f, d) \sqsubseteq \mathsf{apply}(g, d)$ then $f \sqsubseteq g$.*

Proof. Let $f, g \in \mathbb{D}^{\sigma\to\tau}$ such that $f \not\sqsubseteq g$. We will prove $\mathsf{apply}(f, d) \not\sqsubseteq \mathsf{apply}(g, d)$ as follows. By Lemma 4.7, there must be a query (path) q' such that $f' = f@q'$ and $g' = g@q'$ are defined, $f' \neq \bot$, and f' and g' are immediately incomparable. Let d_1 be $\bigsqcup \widehat{q'}(1)$. By the Lemma 4.14, we know that for all $d \sqsupseteq d_1$

$$\mathsf{apply}(f, d)@\mathit{shift}_1(q') = \mathsf{apply}(f', d)$$

and

$$\mathsf{apply}(g, d)@\mathit{shift}_1(q') = \mathsf{apply}(g', d)\,.$$

If we can construct $d \sqsupseteq d_1$ such that

$$\mathsf{apply}(f', d) \not\sqsubseteq \mathsf{apply}(g', d)\,,$$

the lemma immediately follows because @ is monotonic.

We will construct the tree d by performing a case analysis on the form of f' and g':

1. $f', g' \in \mathbb{N}_\bot^{\mathbb{E}}$: The two functions are constant functions returning *different* constants. Let $d = d_1$. Then
$$\mathsf{apply}(f', d) = f' \not\sqsubseteq g' = \mathsf{apply}(g', d)$$
by assumption.

2. $f' \in \mathbb{N}^{\mathbb{E}}_{\perp}, g' = \langle i, p, g'' \rangle$: One function is a constant function and the other inspects the ith argument. If $i > 1$, we let $d = d_1$, and we trivially have

$$\mathsf{apply}(f', d) = f' \not\sqsubseteq \langle i-1, p, \underline{\lambda} x \,.\, \mathsf{apply}(g''(x), d) \rangle = \mathsf{apply}(g', d).$$

   Otherwise $i = 1$, and we distinguish two possible cases:

   (a) $f' \neq \mathsf{error}_1$: Let $d = p[?/\mathsf{error}_1] \sqcup d_1$, which exists since $p \in \mathcal{Q}(\widehat{q'}(1))$ and, in particular, $d_1 @ p = \perp$. It immediately follows that $d @ p = \mathsf{error}_1$ and, since by assumption$f' \neq \perp$,

$$\mathsf{apply}(f', d) = f' \not\sqsubseteq \mathsf{error}_1 = \mathsf{apply}(g', d).$$

   (b) $f' = \mathsf{error}_1$: Let $d = p[?/\mathsf{error}_2] \sqcup d_1$. The rest follows as in the previous subcase.

3. $f' = \langle i, p, f'' \rangle, g' \in \mathbb{N}^{\mathbb{E}}_{\perp}$: This case is symmetric to the preceding one.

4. $f' = \langle i, p, f'' \rangle, g' = \langle j, q, g'' \rangle, i \neq j$: Both functions inspect one of their arguments but different ones, and thus f' cannot approximate g'. Now we must distinguish three subcases, depending on which arguments the functions inspect:

   (a) $i = 1$: Let $d = p[?/\mathsf{error}_1] \sqcup d_1$, which is well-defined by the same argument as in the preceding case. Then, since $d @ p = \mathsf{error}_1$,

$$\mathsf{apply}(f', d) = \mathsf{error}_1 \not\sqsubseteq \langle j-1, q, \underline{\lambda} x \,.\, \mathsf{apply}(g''(x), d) \rangle = \mathsf{apply}(g', d).$$

   (b) $j = 1$: Let $d = q[?/\mathsf{error}_1] \sqcup d_1$. The rest is analogous to the preceding sub-case.

   (c) $i, j > 1$: Let $d = d_1$. The two results differ because the applications yield functions that again inspect different arguments:

$$\begin{array}{rcl} \mathsf{apply}(f', d) & = & \langle i-1, p, \underline{\lambda} x \,.\, \mathsf{apply}(f''(x), d) \rangle \\ & \not\sqsubseteq & \langle j-1, q, \underline{\lambda} x \,.\, \mathsf{apply}(g''(x), d) \rangle \\ & = & \mathsf{apply}(g', d) \end{array}$$

5. $f' = \langle i, p, f'' \rangle, g' = \langle i, q, g'' \rangle, p \neq q$: The two functions inspect the same argument with different queries. We distinguish two cases:

   (a) $i = 1$: Since $q, p \in \mathcal{Q}(\widehat{q'}(1))$, i.e. $d_1 @ q = \perp$, $d_1 @ p = \perp$, and $q \neq p$, the domain elements $q[?/\mathsf{error}_1]$ and $p[?/\mathsf{error}_2]$ are consistent with each other and are consistent with d_1. Hence, let $d = q[?/\mathsf{error}_1] \sqcup p[?/\mathsf{error}_2] \sqcup d_1$. Clearly, $d @ q = \mathsf{error}_1$, $d @ p = \mathsf{error}_2$, and therefore,

$$\mathsf{apply}(f', d) = \mathsf{error}_1 \not\sqsubseteq \mathsf{error}_2 = \mathsf{apply}(g', d)$$

   (b) $i > 1$: Let $d = d_1$:

$$\begin{array}{rcl} \mathsf{apply}(f', d) & = & \langle i-1, p, \underline{\lambda} x \,.\, \mathsf{apply}(f''(x), d) \rangle \\ & \not\sqsubseteq & \\ \mathsf{apply}(g', d) & = & \langle i-1, q, \underline{\lambda} x \,.\, \mathsf{apply}(g''(x), d) \rangle \end{array}$$

The proof is complete since the preceding analysis covers all the possible ways to pick immediately incomparable f' and g' where $f' \not\sqsubseteq g'$. ∎

The extensionality theorem implies that trees behave like mathematical functions under application. Indeed, based on apply, we can define a domain of functions from $\mathbb{T}^\sigma$ to $\mathbb{T}^\tau$ that we can use instead of the tree domain.

Definition 4.17. ($\mathbb{F}^{\sigma\to\tau}$) For each f in $\mathbb{T}^{\sigma\to\tau}$, let $fun(f)$ denote the function $\underline{\lambda}x : \mathbb{T}^\sigma . \mathsf{apply}(f, x)$. The domain $\mathbb{F}^{\sigma\to\tau}$ is the set $\{fun(f) \mid f \in \mathbb{T}^{\sigma\to\tau}\}$ under the pointwise ordering on functions: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x in $\mathbb{T}^\sigma$. ∎

It is straightforward to show that the function domain $\mathbb{F}^{\sigma\to\tau}$ is isomorphic to the tree domain $\mathbb{T}^{\sigma\to\tau}$.

Corollary 4.18 *$\mathbb{T}^{\sigma\to\tau}$ is isomorphic to $\mathbb{F}^{\sigma\to\tau}$.*

Proof. Let $f' = fun(f), g' = fun(g)$. If $f \sqsubseteq g$ then $\mathsf{apply}(f, x) \sqsubseteq \mathsf{apply}(g, x)$ for all x, and therefore $(\underline{\lambda}x . \mathsf{apply}(f, x))(x) \sqsubseteq (\underline{\lambda}x . \mathsf{apply}(g, x))(x)$, which is what we claimed.

Conversely, let $f \in \mathbb{T}^{\sigma\to\tau}$ be some pre-image of $f' \in \mathbb{F}^{\sigma\to\tau}$, and let $g \in \mathbb{T}^{\sigma\to\tau}$ be some pre-image of $g' \in \mathbb{F}^{\sigma\to\tau}$. If $f' \sqsubseteq g'$ then $(\underline{\lambda}x . \mathsf{apply}(f, x))(x) \sqsubseteq (\underline{\lambda}x . \mathsf{apply}(g, x))(x)$ since f and g are pre-images. Hence, $f \sqsubseteq g$ by extensionality.

Clearly, *fun* is onto and it is also easy to check that it is one-to-one. Let $f \in \mathbb{T}^{\sigma\to\tau}$ and $g \in \mathbb{T}^{\sigma\to\tau}$ be two pre-images of $f' \in \mathbb{F}^{\sigma\to\tau}$. It immediately follows that $f \sqsubseteq g$ and $g \sqsubseteq f$, which implies $f = g$. ∎

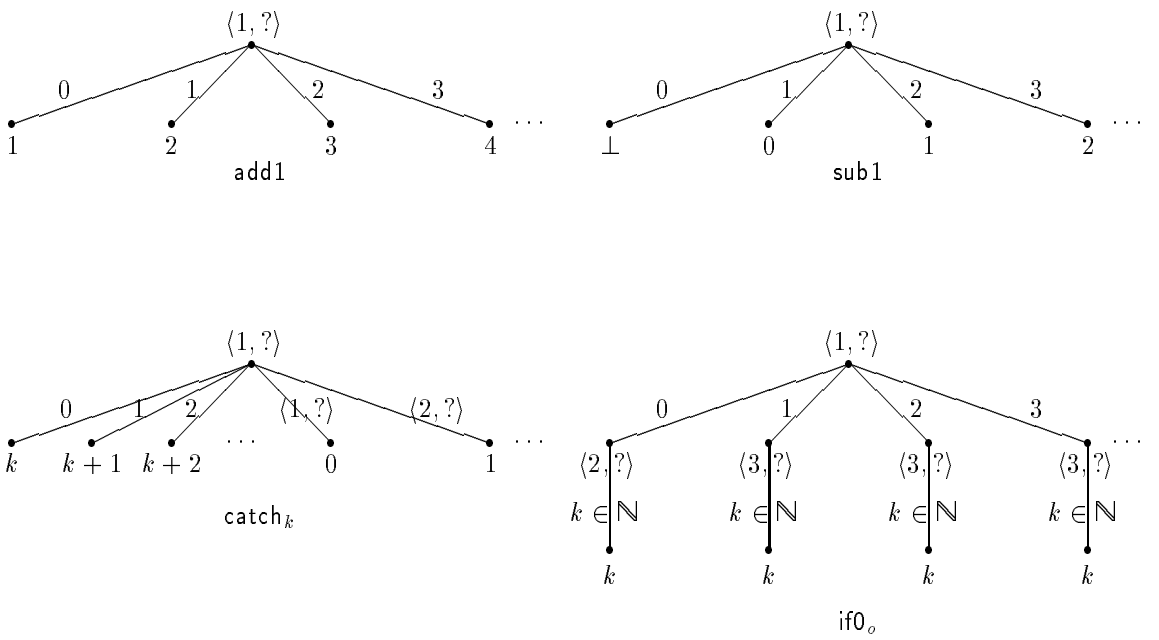


Figure 4: Meanings of Primitive Constants

4.3 Assigning Meaning to Phrases

To assign meaning to phrases in SPCF, we translate programs into a combinatory language in which λ-abstractions are replaced by applications of the combinators $\mathsf{S}^{\sigma,\tau,\upsilon}$, $\mathsf{K}^{\sigma,\tau}$, and I^{σ}:

$$\begin{array}{rcl} M & ::= & x^{\sigma} \mid \ulcorner n \urcorner \mid \mathsf{add1} \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{Y}^{\sigma} \mid \mathsf{catch}^{\sigma} \mid \mathsf{error}_1 \mid \mathsf{error}_2 \mid \\ & & \mathsf{S}^{\sigma,\tau,\upsilon} \mid \mathsf{K}^{\sigma,\tau} \mid \mathsf{I}^{\sigma} \mid \mathsf{apply}^{\sigma,\tau}(M,M) \end{array}$$

The translation is based on the conventional bracket elimination algorithm (see Figure 1). For convenience, we will use multi-ary abstractions and applications as abbreviations for sequences of unary applications:

$$\begin{array}{rcl} \mathsf{apply}(f, x_1, \ldots, x_n) & \stackrel{df}{=} & \mathsf{apply}(\ldots(\mathsf{apply}(f, x_1), \ldots), x_n) \\ \lambda^* x_1, \ldots, x_n \,.\, M & \stackrel{df}{=} & \lambda^* x_1 \,.\, (\ldots(\lambda^* x_n \,.\, M)) \,. \end{array}$$

For any combinatory term M, we will let $M[x/N]$ denote the result of substituting all occurrences of x by the term N.

The bracket elimination algorithm converts every program phrase to a combinatory term in SPCF. These terms are composed solely from variables, constants, and the binary function symbols $\mathsf{apply}^{\sigma,\tau}$. Hence, we can assign meaning to any program by defining the meaning of the constants $\{\ulcorner n \urcorner\} \cup \{\mathsf{add1}, \mathsf{sub1}, \mathsf{if0}, \mathsf{catch}^{\sigma}, \mathsf{error}_1, \mathsf{error}_2, \mathsf{S}^{\sigma,\tau,\upsilon}, \mathsf{K}^{\sigma,\tau}, \mathsf{I}^{\sigma}, \mathsf{Y}^{\sigma}\}$ and the binary function symbols $\mathsf{apply}^{\sigma,\tau}$.

In the sequel, we will say that an expression M denotes the tree d in the model $\mathcal{M}$ if $\mathcal{M}[M] = d$. In equations, we use expressions from SPCF on both sides. Thus, M denotes the final answer a is equivalent to the equation $M = \ulcorner a \urcorner$.

With the exception of fixed point operator Y^{σ}, the primitive constants of SPCF are easy to define in $\mathcal{T}$.

Definition 4.19. ($\mathsf{add1}$, $\mathsf{sub1}$, $\mathsf{if0}$, catch) In the model $\mathcal{T}$, the constants $\mathsf{add1}$, $\mathsf{sub1}$, $\mathsf{if0}$, catch denote the following trees:

$$\begin{array}{rcl} \mathcal{T}[\![\mathsf{add1}]\!] & = & \langle 1, ?, \{(n, n+1) \mid n \in \mathbb{N}\}\rangle \\ \mathcal{T}[\![\mathsf{sub1}]\!] & = & \langle 1, ?, \{(n+1, n) \mid n \in \mathbb{N}\} \cup \{(0, \bot)\}\rangle \\ \mathcal{T}[\![\mathsf{if0}_o]\!] & = & \langle 1, ?, \{(0, \langle 2, ?, (\underline{\lambda} j : \mathbb{N} \,.\, j)\rangle)\} \cup \{(n+1, \langle 3, ?, (\underline{\lambda} j : \mathbb{N} \,.\, j)\rangle) \mid n \in \mathbb{N}\}\rangle \\ \mathcal{T}[\![\mathsf{catch}_k]\!] & = & \langle 1, ?, \{(j, j+k) \mid j \in \mathbb{N}\} \cup \{(\langle j, ?\rangle, j-1) \mid 1 \leq j \leq k\}\rangle \end{array}$$

∎

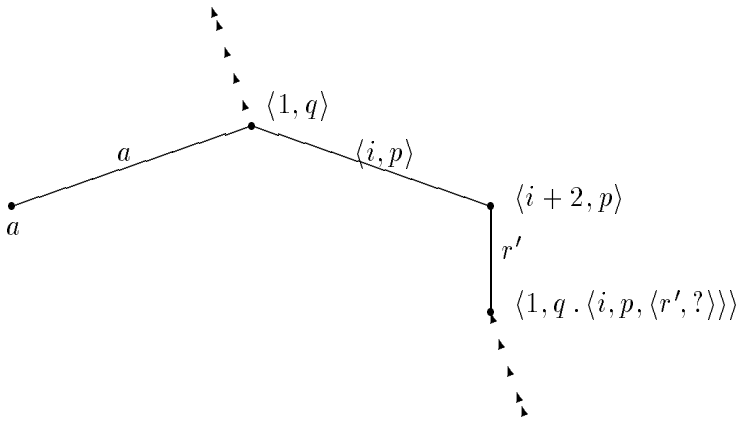
Figure 4 contains pictorial descriptions of the primitive constants other than Y^{σ}. Since the definition of $\mathcal{T}[\![\mathsf{Y}^{\sigma}]\!]$ relies on the meanings of the combinators $\mathcal{O}$ in $\mathcal{T}$, we will defer defining $\mathcal{T}[\![\mathsf{Y}^{\sigma}]\!]$ until the end of this subsection.

The construction of the trees representing the combinators $\mathsf{S}^{\sigma,\tau,\upsilon}$, $K^{\sigma,\tau}$, and I^{σ} is tedious. We begin with the combinator K. Recall that K must satisfy the following equation:

$$\mathsf{apply}(\mathsf{K}, x_1, x_2) = x_1$$

for all x_1 and x_2 in the appropriate domains. Therefore, $\mathsf{apply}(\mathsf{K}, x_1)$ must be a tree that ignores its first argument and returns the tree x_1. In other words, the tree $\mathsf{apply}(\mathsf{K}, x_1)$ does not contain any node values of the form $\langle 1, q\rangle$; it is identical to x_1 except that it contains node values of the form $\langle i+1, q\rangle$ in place of node values of the form $\langle i, q\rangle$. When the tree $\mathsf{apply}(\mathsf{K}, x_1)$ is applied to another tree d, apply traverses $\mathsf{apply}(\mathsf{K}, x_1)$ and decrements every index at every node; d is ignored.

To "copy" the first input with incremented indices, K must completely explore its first argument x_1. It begins this process by probing the root of x_1. If x_1 returns a constant answer, K returns this answer as its own final answer. If x_1 responds with a query q about its ith argument, K queries its $(i+2)$nd argument x_{i+2}, ignoring its second argument x_2. K probes argument position $i+2$ because it must skip over two arguments, x_1 and x_2 to reach the first argument for x_1. The answer of x_{i+2} to K's query is the information that x_1 requested. Thus, after obtaining a response from x_{i+2}, K probes x_1 again with a new query that extends x_1's response with the additional information provided by x_{i+2}. The subsequent response by x_1 is subjected to the same case analysis as the first one. Pictorially, K has the following structure at every layer of the tree:



The preceding discussion demonstrates that the tree representing K encodes a recursive search process. This process is formalized in the following definition.

Definition 4.20. (K) In the model $\mathcal{T}$, the combinator $\mathsf{K}^{\sigma,\tau}$ denotes the tree $K(?)$ where K is a recursive function mapping queries in $\mathbb{Q}^\sigma$ to subtrees in $\mathbb{T}^{\sigma\to\tau}(\widehat{q})$ defined by the equation:

$$K(q) = \langle 1, q, \underline{\lambda} r \,.\, \begin{cases} a & r = q[?/a], a \in \mathbb{N} \\ \langle i+2, p, \underline{\lambda} r' \,.\, K(r \,.\, \langle r', ?\rangle)\rangle & r = q[?/\langle i, p\rangle] \end{cases} \rangle$$

where $\mathbb{T}^{\sigma\to\tau}(\widehat{q})$ denotes the domain determined by the finitary basis $\mathbb{D}^{\sigma\to\tau}(\widehat{q})$. The solution of this recursion equation is the least-upper bound of finitely deep approximations to K (which map $\mathbb{Q}^\sigma$ to $\mathbb{D}^{\sigma\to\tau}(\widehat{q})$):

$$K(q) = \bigsqcup \{K_n(q) \mid n \in \mathbb{N}\}$$

where

$$K_0(q) \;=\; \bot$$

$$K_{n+1}(q) \;=\; \langle 1, q, \underline{\lambda} r \,.\, \begin{cases} a & r = q[?/a], a \in \mathbb{N} \\ \langle i+2, p, \underline{\lambda} r'.K_n(r \,.\, \langle r', ?\rangle)\rangle & r = q[?/\langle i, p\rangle] \end{cases} \rangle$$

It is easy to check that

$$\mathsf{K} = \bigsqcup \{K_n(?) \mid n \in \mathbb{N}\} \,.$$

■

The definition of the I combinator is similar to the definition of K; the details are omitted.

The S combinator is most complex primitive operation in $\mathcal{T}$. It is the least solution to the equation

$$\begin{aligned} \mathsf{apply}(\mathsf{S}, x_1, x_2, x_3) &\stackrel{df}{=} \mathsf{apply}(\mathsf{apply}(\mathsf{apply}(\mathsf{S}, x_1), x_2), x_3) \\ &= \mathsf{apply}(\mathsf{apply}(x_1, x_3), \mathsf{apply}(x_2, x_3)) \\ &= \mathsf{apply}(x_1, x_3, \mathsf{apply}(x_2, x_3)) \,. \end{aligned}$$

Like K, S probes its first argument with progressively deeper queries q until it gets a final answer. Intermediate responses from x_1 are queries about its arguments in the application

$$\mathsf{apply}(x_1, x_3, \mathsf{apply}(x_2, x_3))$$

namely x_3, $\mathsf{apply}(x_2, x_3)$, x_4, x_5, ... To define S, we must distinguish the following responses r by x_1 to the query q:

1. $r = q[?/a]$: If x_1 returns a final answer, S returns the same final answer.

2. $r = q[?/\langle i+2, p\rangle]$: If x_1's response is a query p about its $(i+2)$nd argument, S must probe position p in x_{i+3}.

3. $r = q[?/\langle 1, p\rangle]$: If x_1 asks for a node value from its first argument x_3, S must provide the requested information. But S may not have to query x_3 to get this node value because S also gathers information about x_3 when x_1 asks for a node value from its second argument $\mathsf{apply}(x_2, x_3)$. Thus, S may have already extracted the information in handling previous responses from x_1 of the form $q[?/\langle 2, p\rangle]$. Since S must not generate the same query twice, we must maintain a finite tree d_3 recording the information extracted from x_3 at each stage in the construction of S. We will use this "table" to avoid inserting redundant queries in S. At position q' in S, d_3 is simply $\widehat{q'}(3)$. If S has already extracted the information requested by x_1 (*i.e.* $\overline{p} \sqsubset d_3$), then S probes x_1 again with the query q extended by $d_3@p$. Otherwise, S must probe x_3 for the node value at position p before it probes x_1 again.

4. $r = q[?/\langle 2, p\rangle]$: If x_1 probes its second argument, S must simulate the application of x_2 to x_3. The simulation must incorporate the information in the table d_3 to avoid generating redundant queries of x_3. Since S must perform this simulation incrementally, we will also maintain a finite tree d_2 that records the approximation that S has accumulated for x_2 at each position q' in S. The finite tree d_2 has an unusual form. Every node of the form $\langle 1, p, f\rangle$ has exactly one son below it, *i.e.*, the proper domain of f contains exactly one element, because apply only explores x_3's response to this

query. The other sons in x_2 below this node correspond to other values of x_3! To compute the node value at position p in the tree $\mathsf{apply}(x_2, x_3)$, S must simulate the construction of the specified node. By examining d_2, S can determine which node in x_2 should be applied to x_3, resuming the simulation of $\mathsf{apply}(x_2, x_3)$ at the appropriate point. We will encapsulate the analysis of d_2 in an auxiliary function *encode* that maps d_2 and p to the appropriate query about x_2, which must be a member of the set $\mathcal{Q}(\{d_2\})$. We will defer further discussion of *encode* until we complete our analysis of the construction of S. When S probes x_2 with the query $\mathit{encode}(d_2, p)$, x_2 can produce three different kinds of responses r':

(a) $r' = q[?/a]$: If x_2 returns a final answer, S may resume the process of probing x_1 with the knowledge that $\mathsf{apply}(x_2, x_3)@p = a$.

(b) $r' = q[?/\langle 1, p'\rangle]$: If x_2 returns a query p' about its first argument, S must redirect this query to x_3 if the information is not already available in the table d_3. If $\overline{p'} \sqsubset d_3$, then S already knows the node value of x_3 at position p' and continues probing x_2 to simulate the application of x_2 to x_3. On the other hand, if $\overline{p'} \not\sqsubset d_3$, S must probe x_3 with query p' to obtain the information requested by x_2 before it continues probing x_2. If S probes x_3, it must update the table d_3.

(c) $r' = q[?/\langle i+1, p'\rangle]$: If x_2 returns a query p' about an argument with index greater than 1, this query appears in shifted form as a node value in $\mathsf{apply}(x_2, x_3)$. Similary, all of the nodes above this node in $\mathsf{apply}(x_2, x_3)$ are shifted copies of the node values in r'; each node of the form $\langle j+1, q'\rangle$ in x_2 corresponds to a node of the form $\langle j, q'\rangle$ in $\mathsf{apply}(x_2, x_3)$. S must resume probing x_1 using a shifted form of the query r'. The function shift_1 defined in the preceding section (Definition 4.13) performs precisely this shifting transformation.

To complete our informal discussion of how to construct the tree representing S, we need to describe how the function *encode* works. Given inputs d_2 (the current approximation to x_2) and p (the query by x_1), the encoding function *encode* traverses d_2 and p to discover which node in x_2 must be examined next to determine the node value at the root of the subtree $\mathsf{apply}(x_2, x_3)@p$. Much of the information in d_2 and p is redundant because either (i) $d_2 = \bot$ and $p = ?$ or (ii) $p = \mathit{shift}_1(d_2) \,.\, \langle r, ?\rangle$ for some response r. To identify this node position in x_2, *encode* must translate p to the corresponding path p' in x_2. For every value $\langle i, q\rangle$ in p, there is a corresponding node $\langle i+1, q\rangle$ in p'. The path p', however, may contain more nodes than p because node values in p' of the form $\langle 1, q\rangle$ in x_2 are eliminated in the formation of the tree $\mathsf{apply}(x_2, x_3)$. Consequently, *encode* must insert these node values in p' using the information stored in d_2.

We can formalize the preceding description of S by defining a tree generating function S that calls an auxiliary tree generating function T and an encoding function *encode*. The function S generates the portions of the tree that encode the query process that S performs on x_1. The inputs to S are finite elements approximating x_2 and x_3 and a query q about x_1. The T function generates the parts of the tree that encode the query process on x_2. It uses finite elements that approximate x_2 and x_3, the last response by x_1, and a query about x_2. The *encode* function translates queries by x_1 about its second argument $\mathsf{apply}(x_2, x_3)$ into corresponding queries about x_2. It takes a finite element approximating x_2 and a query about $\mathsf{apply}(x_2, x_3)$ as inputs.

Definition 4.21. (S) In the model $\mathcal{T}$, the combinator $\mathsf{S}^{\sigma,\tau,\upsilon}$ denotes the tree $S(?,?)$ where

$$S : \mathbb{Q}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon} \times \mathbb{Q}^{\sigma\to\tau\to\upsilon} \longrightarrow \bigcup_{\rho} \mathbb{T}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\rho)$$

is defined by the recursion equations in Figure 5. The figure also gives definitions for the auxiliary functions

$$T : \mathbb{R}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon} \times \mathbb{Q}^{(\sigma\to\tau)} \longrightarrow \bigcup_{\rho} \mathbb{T}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\rho)$$

and $encode : \mathbb{D}^{\sigma\to\tau} \times \mathbb{Q}^{\tau} \longrightarrow \mathbb{Q}^{\sigma\to\tau}$.

$$
\begin{array}{l}
S(p_S, q_1) \;=\; \\
\quad \mathsf{let}\ \{d_2 = \widehat{p_S}(2);\ d_3 = \widehat{p_S}(3)\} \\
\quad \mathsf{in} \\
\qquad \langle 1, q_1, \underline{\lambda} r_1 \in \mathcal{R}(q_1)\,.\, \left\{ \begin{array}{ll}
a & r_1 = q_1[?/a] \\
S(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], & r_1 = q_1[?/\langle 1, p\rangle], \\
\quad r_1\,.\,\langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle) & d_3@p \neq \bot \\
\langle 3, p, \underline{\lambda} s \in \mathcal{R}(p)\,. & r_1 = q_1[?/\langle 1, p\rangle], \\
\quad S(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], & d_3@p = \bot \\
\qquad r_1\,.\,\langle s, ?\rangle)\rangle & \\
T(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], encode(d_2, p)) & r_1 = q_1[?/\langle 2, p\rangle] \\
\langle i+3, p, \underline{\lambda} s \in \mathcal{R}(p)\,. & r_1 = q_1[?/\langle i+2, p\rangle] \\
\quad S(p_S[?/\langle 1, q_1, \langle r_1, \langle i+3, p, \langle s, ?\rangle\rangle\rangle\rangle], & \\
\qquad r_1\,.\,\langle s, ?\rangle)\rangle &
\end{array} \right. \rangle \\
T(p_S, q_2) \;=\; \\
\quad \mathsf{let}^*\ \{r_1 = \widehat{p_S}(1);\ d_2 = \widehat{p_S}(2);\ d_3 = \widehat{p_S}(3);\ q_1[?/\langle 2, p\rangle] = r_1\} \\
\quad \mathsf{in} \\
\qquad \langle 2, q_2, \underline{\lambda} r_2 \in \mathcal{R}(q_2)\,. \left\{ \begin{array}{ll}
S(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle],\ r_1\,.\,\langle shift_1(r_2), ?\rangle) & r_2 = q_2[?/a] \\
T(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], & r_2 = q_2[?/\langle 1, p\rangle], \\
\quad r_2\,.\,\langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle) & d_3@p \neq \bot \\
\langle 3, p, \underline{\lambda} s \in \mathcal{R}(p)\,. & r_2 = q_2[?/\langle 1, p\rangle], \\
\quad T(p_S[?/\langle 2, q_2, \langle r_2, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], & d_3@p = \bot \\
\qquad r_2\,.\,\langle s, ?\rangle)\rangle & \\
S(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle],\ r_1\,.\,\langle shift_1(r_2), ?\rangle) & r_2 = q_2[?/\langle i+1, p\rangle]
\end{array} \right. \rangle
\end{array}
$$

$$
\begin{array}{rcl}
encode(\bot, ?) & = & ? \\
encode(\langle 1, q, \langle r, d_2'\rangle\rangle, p) & = & \langle 1, q, encode(d_2', p)\rangle \\
encode(\langle i+1, q, f\rangle, \langle i, q, \langle r, p'\rangle\rangle) & = & \langle i+1, q, encode(f(r), p')\rangle \\
encode(\langle i+1, q\rangle, \langle i, q, \langle r, ?\rangle\rangle) & = & \langle i+1, q, \langle r, ?\rangle\rangle
\end{array}
$$

Figure 5: Definition of the functions S, T, and $encode$

A solution to the recursion equations in Figure 5 can be obtained by defining an infinite series of finitely deep approximations as in the definition of K above. The pattern matching notation in the definition of the *encode* function relies on two facts:

1. The approximation d_2 has exactly one son below each node value of the form $\langle 1, p\rangle$.
2. Every invocation of the definition $encode(d_2, p)$ satisfies the invariant: there exists a query q such that $d_2@q = \bot$ and $p = shift_1(q)$.

When the preceding two conditions hold, it is easy to prove that $encode(d_2, p) = q$ by induction on q. Appendix A contains a proof that the decision-tree $\mathcal{T}[\![\mathsf{S}]\!]$ is well-defined. ∎

With the definitions of $\mathcal{T}[\![\mathsf{K}]\!]$ and $\mathcal{T}[\![\mathsf{S}]\!]$ in hand, we are finally ready to define the meaning of the constant Y^σ in $\mathcal{T}$. The meaning $\mathcal{T}[\![\mathsf{Y}^\sigma]\!]$ must satisfy the equation

$$\mathcal{T}[\![\mathsf{apply}(M, \mathsf{apply}(\mathsf{Y}^\sigma, M))]\!] = \mathcal{T}[\![\mathsf{apply}(\mathsf{Y}^\sigma, M)]\!]$$

for all closed combinatory terms M of type $\sigma \to \sigma$. Since the tree domain $\mathbb{T}^{\sigma\to\sigma}$ is isomorphic to the function domain $\mathbb{F}^{\sigma\to\sigma}$ and the functions in $\mathbb{F}^{\sigma\to\sigma}$ are continuous, we can exploit the usual limiting construction to define $\mathcal{T}[\![\mathsf{Y}^\sigma]\!]$. For the duration of this construction, let Ω^τ for $\tau \in \mathit{Type}$ abbreviate the expression given by the following inductive definition:

$$\begin{aligned} \Omega^o &= \mathsf{apply}(\mathsf{sub1}, 0) \\ \Omega^{\tau\to\upsilon} &= \lambda^* x^\tau \,.\, \Omega^\upsilon \,. \end{aligned}$$

The expression Ω^τ has the same meaning ($\bot_\tau$) in $\mathcal{T}$ as the definition for Ω^τ given at the beginning of Section 2, but the new definition does not contain any occurrences of constants Y^τ. Given the definition of Ω^τ, we define

$$\mathcal{T}[\![\mathsf{Y}^\sigma]\!] = \bigsqcup \{\mathcal{T}[\![\lambda^* f \,.\, \underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}]\!] \mid n \in \mathbb{N}\} \,.$$

It is straightforward to prove that $\mathcal{T}[\![\mathsf{Y}^\sigma]\!]$ is well-defined. First, the combinatory term

$$\lambda^* f \,.\, \underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}$$

does not contain any constants Y^τ, so

$$\mathcal{T}[\![\lambda^* f \,.\, \underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(x, \Omega^\sigma)\ldots)}_{n}]\!]$$

is well-defined for all $n \geq 0$. Second, it is easy to prove that the set

$$\{\mathcal{T}[\![\lambda^* f \,.\, \underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^{\sigma\to\sigma})\ldots)}_{n}]\!] \mid n \in \mathbb{N}\}$$

forms a chain by induction on n (since $\mathcal{T}[\![\mathsf{apply}]\!]$ is monotonic). Hence, the specified least upper bound exists. We will prove that $\mathcal{T}[\![\mathsf{Y}^\sigma]\!]$ is the tree encoding the least fixed point operator of type $(\sigma \to \sigma) \to \sigma$ in the next subsection.

4.4 Operational Properties of the Model

To provide some operational insight about the model and its combinators, we derive some equational laws that the model $\mathcal{T}$ satisfies. The most important properties are the equations asserting that the combinators S, K, and I satisfy their definitions.

Theorem 4.22 *Let x, y, z be variables ranging over arbitrary elements in appropriate domains. Then:*

$$\mathsf{apply}(\mathsf{S}, x, y, z) = \mathsf{apply}(\mathsf{apply}(x, z), \mathsf{apply}(y, z)) \qquad (\mathbf{S})$$

$$\mathsf{apply}(\mathsf{K}, x, y) = x \qquad (\mathbf{K})$$

$$\mathsf{apply}(\mathsf{I}, x) = x \qquad (\mathbf{I})$$

Proof. The proofs for S and K are given in Appendix A. The proof for I closely follows the proof for K. ∎

The preceding theorem implies that the β axiom holds for combinatory terms translated from λ-abstractions [1]. In addition, the extensionality theorem (Theorem 4.11) and the validity of β axiom imply that the η axiom also holds.

Corollary 4.23 (β, η) *Let M and N be combinatory terms.*

(i) *If $\mathcal{E}$ is an environment that binds each variable x^τ in $FV(M) \cup FV(N)$ to an element in $\mathbb{T}^\tau$, then*

$$\mathcal{T}[\![\mathsf{apply}(\lambda^* y\,.\,M, N)]\!]_{\mathcal{E}} = \mathcal{T}[\![M[y/N]]\!]_{\mathcal{E}}\,. \qquad (\beta)$$

(ii) *If $\mathcal{E}$ is an environment that binds each variable x^τ in $FV(M)$ to an element in $\mathbb{T}^\tau$, then:*

$$\mathcal{T}[\![\lambda^* y\,.\,\mathsf{apply}(M, y)]\!]_{\mathcal{E}'} = \mathcal{T}[\![M]\!]_{\mathcal{E}'} \qquad (\text{if } y \notin FV(M)) \qquad (\eta)$$

Proof. Both equations can be proved using standard methods [1]; the proof of (η) depends on the extensionality theorem (Theorem 4.11). ∎

We can use the β axiom to prove that $\mathcal{T}[\![\mathsf{Y}]\!]$ is the tree encoding the least fixed point operator.

Corollary 4.24 (Y operator) *For all closed combinatory terms M of type $\sigma \to \sigma$,*

$$\mathcal{T}[\![\mathsf{apply}(M, \mathsf{apply}(\mathsf{Y}^\sigma, M))]\!] = \mathcal{T}[\![\mathsf{apply}(\mathsf{Y}^\sigma, M)]\!]\,.$$

Proof. Let the expression Ω^τ be defined as in Section 4.4 above. Since the function apply is continuous for any tree $g \in \mathbb{T}^{\sigma\to\sigma}$,

$$
\begin{aligned}
&\mathcal{T}[\![\mathsf{apply}(M, \mathsf{apply}(\mathsf{Y}^\sigma, M))]\!] \\
&\quad = \mathsf{apply}(\mathcal{T}[\![M]\!], \mathsf{apply}(\mathcal{T}[\![\mathsf{Y}^\sigma]\!], \mathcal{T}[\![M]\!])) \\
&\quad = \mathsf{apply}(\mathcal{T}[\![M]\!], \mathsf{apply}(\bigsqcup\{\mathcal{T}[\![\lambda^* f\,.\,\underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}]\!] \mid n \in \mathbb{N}\}, \mathcal{T}[\![M]\!])) \\
&\quad = \bigsqcup\{\mathsf{apply}(\mathcal{T}[\![M]\!], \mathcal{T}[\![\mathsf{apply}(\lambda^* f\,.\,\underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}, M)]\!]) \mid n \in \mathbb{N}\} \\
&\quad = \bigsqcup\{\mathcal{T}[\![\mathsf{apply}(M, \mathsf{apply}(\underbrace{\mathsf{apply}(M, \ldots, \mathsf{apply}(M, \Omega^\sigma)\ldots)}_{n}, M))]\!] \mid n \in \mathbb{N}\} \\
&\quad = \bigsqcup\{\mathcal{T}[\![\mathsf{apply}(M, \mathsf{apply}(\underbrace{\mathsf{apply}(M, \ldots, \mathsf{apply}(M, \Omega^\sigma)\ldots)}_{n}, M))]\!] \mid n \in \mathbb{N}\} \\
&\quad = \bigsqcup\{\mathcal{T}[\![\mathsf{apply}(\lambda^* f\,.\,\mathsf{apply}(f, \mathsf{apply}(\underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}, f)), M)]\!] \mid n \in \mathbb{N}\} \\
&\quad = \bigsqcup\{\mathsf{apply}(\mathcal{T}[\![\lambda^* f\,.\,\mathsf{apply}(f, \mathsf{apply}(\underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}, f))]\!], M) \mid n \in \mathbb{N}\} \\
&\quad = \mathsf{apply}(\bigsqcup\{\mathcal{T}[\![\lambda^* f\,.\,\mathsf{apply}(f, \mathsf{apply}(\underbrace{\mathsf{apply}(f, \ldots, \mathsf{apply}(f, \Omega^\sigma)\ldots)}_{n}, f))]\!] \mid n \in \mathbb{N}\}, \mathcal{T}[\![M]\!]) \\
&\quad = \mathsf{apply}(\mathcal{T}[\![\mathsf{Y}^\sigma]\!], \mathcal{T}[\![M]\!]) \\
&\quad = \mathcal{T}[\![\mathsf{apply}(\mathsf{Y}^\sigma, M)]\!]\,. \blacksquare
\end{aligned}
$$

To determine the relevant equations for the catch operators and error values, we draw on our operational intuition about the behavior of control operators [11, 12]. The key idea is to define a "program counter" for program text corresponding to the obvious reduction semantics for SPCF. This counter determines which term in a program must be evaluated next in order to determine the program's answer. In SPCF, the order of evaluation is precisely determined because the primitive operations of SPCF are error-sensitive. We formalize the concept of a textual program counter by defining the notion of *evaluation context*.

Definition 4.25. (*Evaluation context*) An *evaluation context* for SPCF is a syntactic context E that conforms to the following inductive definition (expressed in BNF):

$$
\begin{aligned}
E &::= [\ ] \mid \mathsf{apply}(F, E) \mid \mathsf{apply}(E, M) \mid \mathsf{apply}(\mathsf{catch}, (\lambda^* x_1, \ldots, x_n\,.\,E)) \\
F &::= \mathsf{add1} \mid \mathsf{sub1} \mid \mathsf{if0} \mid \mathsf{catch}
\end{aligned}
$$

∎

An evaluation context is a syntactic context where the hole is in "leftmost-outermost" position. A hole is in "leftmost-outermost" position iff every operator to the left of the the path from the root to the hole is a primitive function; catch operators are classified as primitive functions. If we place a variable x corresponding to a surrounding top-level abstraction in the hole of an evaluation context, the resulting term denotes a function that is strict in the argument x.

Lemma 4.26 *For all variables $x_1, \ldots, x_n$, for all evaluation contexts E, and for all j, $1 \leq j \leq n$,*

$$\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, E[x_j]]\!] = \langle j, ?, f \rangle$$

for some appropriate branching function f,

Proof. See Appendix B. ∎

If we place an error value or $\bot$ in the hole of an an evaluation context, the entire term denotes the value placed in the hole. In addition, catch can determine which variable occurs in the hole of an evaluation context in its procedure argument.

Theorem 4.27 *For all evaluation contexts E, types $\tau = \tau_1 \to \ldots \to \tau_n$, and variables $x_1, \ldots, x_n$,*

$$\begin{aligned}
\mathcal{T}[\![E[\mathsf{error}_j]]\!] &= \mathsf{error}_j && \textbf{(error)}\\
\mathcal{T}[\![E[\bot]]\!] &= \bot && \textbf{(bottom)}\\
\mathcal{T}[\![\mathsf{apply}(\mathsf{catch}^\tau,\, \lambda^* x_1, \ldots, x_n \,.\, E[x_j])]\!] &= \ulcorner j - 1 \urcorner \text{ for } 1 \leq j \leq n && \textbf{(catch)}\\
\mathcal{T}[\![\mathsf{apply}(\mathsf{catch}^\tau,\, \lambda^* x_1, \ldots, x_n \,.\, \ulcorner k \urcorner)]\!] &= \ulcorner k + n \urcorner && \textbf{(return)}
\end{aligned}$$

Proof. By the preceding lemma,

$$\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, E[x_j]]\!] = \langle j, ?, f \rangle$$

for an appropriate branching function f. Thus, for example,

$$\begin{aligned}
\mathcal{T}[\![E[\mathsf{error}_1]]\!] &= \mathcal{T}[\![\mathsf{apply}(\lambda^* x \,.\, E[x], \mathsf{error}_1)]\!] \text{ by } \beta\\
&= \mathsf{apply}(\langle 1, ?, f \rangle, \mathsf{error}_1)\\
&= \mathsf{error}_1
\end{aligned}$$

The other equations follow from similarly simple calculations. ∎

The natural equations for the primitive operations $\{\mathsf{add1}, \mathsf{sub1}, \mathsf{if0}, \mathsf{Y}, \mathsf{error}_1, \mathsf{error}_2\}$, the β and η rules, and the equations for catch completely determine an operational reduction semantics for SPCF [11, 12]. The equations also suffice to prove the representability lemma of the following section.

Note: call/cc$_d$ vs CATCH. As indicated in Section 2, Scheme's original CATCH construct permits the specification of a return value to the "catch label". Since such a control construct is common in many Lisp, Scheme, and ML dialects, the question arises how this construct relates to the less pragmatic, somewhat artificial, catch construct in SPCF. To study this relationship, we introduce the procedure call/cc$_d$ of type $((o \to o) \to o) \to o$ ("call with current (downward) continuation"). This new procedure is similar to Scheme's call/cc procedure. It applies its argument to the current continuation, but unlike full-fledged call/cc-type continuations, call/cc$_d$-continuations can be used at most once. While call/cc$_d$ is a new functional constant, it is easy to see that the binding form of (downward) CATCH and call/cc$_d$ have the same expressive power:

$$\begin{aligned}
(\mathsf{CATCH}\; e\; M) &= (\mathsf{call/cc}_d\; (\lambda e \,.\, M))\\
(\mathsf{call/cc}_d\; f) &= (\mathsf{CATCH}\; e\; (f\; e))\,.
\end{aligned}$$

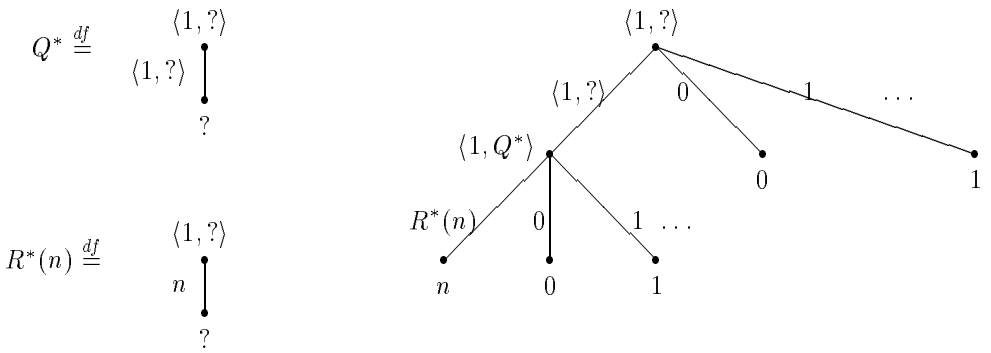


Figure 6: The denotation of $\mathsf{call/cc}_d$

The denotation of $\mathsf{call/cc}_d$ is the tree shown in Figure 6. From an operational perspective, $\mathsf{call/cc}_d$ is defined by the equations [11,12]:

$$\begin{aligned} \mathsf{call/cc}_d(\lambda f^{o \to o} \,.\, E[f \ulcorner n \urcorner]) &= \ulcorner n \urcorner \\ \mathsf{call/cc}_d(\lambda f^{o \to o} \,.\, \ulcorner n \urcorner) &= \ulcorner n \urcorner \end{aligned}$$

where the set of evaluation contexts is adapted *mutatis mutandis*.

With $\mathsf{call/cc}_d$, it is easy to define catch_k for each arity $k \geq 0$:[8]

$$\begin{aligned} \mathsf{catch}_0 &= \lambda x \,.\, x \\ \mathsf{catch}_k &= \lambda f \,.\, \mathsf{call/cc}_d(\lambda g \,.\, (\mathsf{add1}^k \ (f \ (g \ \ulcorner 0 \urcorner) \ldots (g \ \ulcorner k - 1 \urcorner))))). \end{aligned}$$

The converse is more difficult. An application of the form $(\mathsf{call/cc}_d \ f)$ can evaluate the tree representing f to arbitrary depth. To simulate $(\mathsf{call/cc}_d \ f)$, we must write a procedure $test_v$ with free variable v that applies f to a continuation c that determines whether f evaluates c (the continuation) and, if so, whether f passes the value v to c as its parameter. Moreover, the procedure must reveal this information in the root node of its tree representation so that catch can extract it. The following code repeatedly applies catch to the procedure $test_v$ for increasing values $0, 1, \ldots$ of v until v equals the value a that is "thrown" out of the body of f:[9]

$$\begin{aligned} \mathsf{call/cc}_d &= \lambda f \,.\, \mathbf{letrec}\ L = \lambda v \,. \\ &\qquad \mathbf{let}\ test_v = \lambda x y \,.\, (f(\lambda a \,.\, \mathsf{if0}\,(-\ a\ v)\ x\ y))) \end{aligned}$$

[8] The notation $(\mathsf{add1}^k \ x)$ denotes a k-fold application of $\mathsf{add1}$ to x; similarly, $(\mathsf{sub1}^k \ x)$ denotes a k-fold application of $\mathsf{sub1}$ to x.

[9] We are using the abbreviations:

$$\begin{aligned} \mathbf{letrec}\ L = M\ \mathbf{in}\,(L \ \ulcorner 0 \urcorner) &\stackrel{df}{=} (\mathsf{Y}(\lambda L \,.\, M)\ulcorner 0 \urcorner) \\ \mathbf{let}\ x = M\ \mathbf{in}\ N &\stackrel{df}{=} ((\lambda x \,.\, N)M) \end{aligned}$$

$$\begin{array}{l}\quad\textbf{in let } root = (\mathsf{catch}\ test_v)\\ \qquad\textbf{in } (\mathsf{if0}\ root\ a\ (\mathsf{if0}\ (\mathsf{sub1}\ root)\ (L\ (\mathsf{add1}\ v))\ (\mathsf{sub1}^2\ root)))\\ \textbf{in } (L\ \ulcorner 0 \urcorner)\end{array}$$

This code also detects when f is some constant c and returns the same value as f, in this case $(\mathsf{sub1}^2\ root)$.

In summary, $\mathsf{call/cc}_d$ and catch are equally expressive in SPCF. We selected catch as the primitive operation for SPCF for purely technical reasons. Indeed, it is much simpler to program with $\mathsf{call/cc}_d$. **End of Note.**

5 The Full Abstraction Theorem

Plotkin [18] showed that an extensional model of PCF with parallel operations is fully abstract if the language can define all finite elements in the model. Exactly the same argument applies to SPCF. Since we already know that the tree model $\mathcal{T}$ is order-extensional, we can prove that $\mathcal{T}$ is fully abstract by showing that all finite trees are definable in SPCF. In this section, we use this strategy to prove that $\mathcal{T}$ is a fully abstract model of SPCF.

Theorem 5.1 (Full Abstraction of $\mathcal{T}$ and SPCF) *For all SPCF phrases M and N, $M \equiv_{\mathcal{T}} N$ iff $M \simeq N$.*

Proof. The proof follows exactly the same outline as Plotkin's proof that continuous function model is fully abstract for PCF with parallel operations [18:240]. We only need to show that for all phrases M and N that $M \simeq N$ implies $M \equiv_{\mathcal{T}} N$. The other direction is an immediate consequence of the compositionality of the meaning function $\mathcal{T}$. Let M and N be any phrase such that $M \not\equiv_{\mathcal{T}} N$. We will prove that $M \not\simeq N$. Let $\mathcal{E}$ be an environment binding all the variables in $FV(M) \cup FV(N)$ to elements of appropriate type such that

$$\mathcal{T}[\![M]\!]_{\mathcal{E}} \neq \mathcal{T}[\![M]\!]_{\mathcal{E}}\,.$$

If M and N have type o, the proof is trivial: the empty context $[\ ]$ establishes that $M \not\simeq N$ in the environment $\mathcal{E}$. So we can assume that M and N belong to some higher type

$$\tau = \sigma_1 \to \ldots \to \sigma_k \to o\,.$$

Since $\mathcal{T}[\![M]\!]_{\mathcal{E}} \neq \mathcal{T}[\![N]\!]_{\mathcal{E}}$, the domains $\mathbb{T}^{\sigma_i}$ are algebraic, apply is continuous, and SPCF is extensional, it is easy to prove by contradiction that there exist *finite* trees $d_1 \sqsubseteq t_1, \ldots, d_k \sqsubseteq t_k$ such that

$$\mathsf{apply}(\mathcal{T}[\![M]\!]_{\mathcal{E}}, d_1, \ldots, d_k) \neq \mathsf{apply}(\mathcal{T}[\![N]\!]_{\mathcal{E}}, d_1, \ldots, d_k)\,.$$

If we construct closed expressions $E_1, \ldots, E_k$ in SPCF that *represent* the elements $d_1, \ldots, d_k$ (*i.e.* $\mathcal{T}[\![E_i]\!] = d_i, 1 \leq i \leq k$), then we can use the context

$$C[\ ] = (\ [\ ]\ E_1\ \ldots\ E_k)$$

to show that $M \not\simeq N$. In particular,

$$\mathcal{T}[\![C[M]]\!]_{\mathcal{E}} = \mathcal{T}[\![(M\ E_1\ \ldots\ E_k)]\!]_{\mathcal{E}} = \mathsf{apply}(\mathcal{T}[\![M]\!]_{\mathcal{E}}, e_1, \ldots, e_k)$$

while

$$\mathcal{T}[\![C[N]]\!]_{\mathcal{E}} = \mathcal{T}[\![(N\ E_1\ \ldots\ E_k)]\!]_{\mathcal{E}} = \mathsf{apply}(\mathcal{T}[\![N]\!]_{\mathcal{E}}, e_1, \ldots, e_k)\,.$$

Thus, the proof that SPCF is fully abstract reduces to the proof of the following lemma. ∎

Lemma 5.2 *For every finite element $d \in \mathbb{D}^\tau$, there is a closed SPCF expression M such that $\mathcal{T}[\![M]\!] = d$.*

Proof. The proof of the lemma proceeds by induction on the *depth* of the type

$$\tau = \tau_1 \to \ldots \to \tau_k \to o, \quad k \geq 0.$$

The *depth* of ground type o is 1, the *depth* of τ is $1 + \mathsf{max}\{depth(\tau_i) \mid 1 \leq i \leq k\}$. Since the induction step decomposes the tree d into a root node, a set of proper responses, and an attached set of *sub*trees, we need to prove a stronger result than the lemma because *sub*trees of d are not necessarily trees in $\mathbb{D}^\tau$. Specifically, we must prove that for all contexts $\rho \in \mathbb{C}^\tau$, every finite *sub*tree e in $\mathbb{D}^\tau(\rho)$ is "representable" in SPCF. For technical reasons, we will use an equational characterization of subtree representability.

Definition 5.3. (*Subtree representability*) Let ρ be a context in $\mathbb{C}^\tau$. A *subtree* $e \in \mathbb{D}^\tau(\rho)$ is *representable* iff there exists a closed expression M such that

$$\mathsf{apply}(\mathcal{T}[\![M]\!], d_1, \ldots, d_k) = \mathsf{apply}(e, d_1, \ldots, d_k)$$

for all arguments $d_1, \ldots, d_k$ in $\mathbb{D}^{\tau_1}, \ldots, \mathbb{D}^{\tau_k}$, such that $d_i \sqsupseteq \bigsqcup \rho(i)$. ∎

When $\rho = \emptyset$, e is a complete tree and subtree representability reduces to tree representability by extensionality.

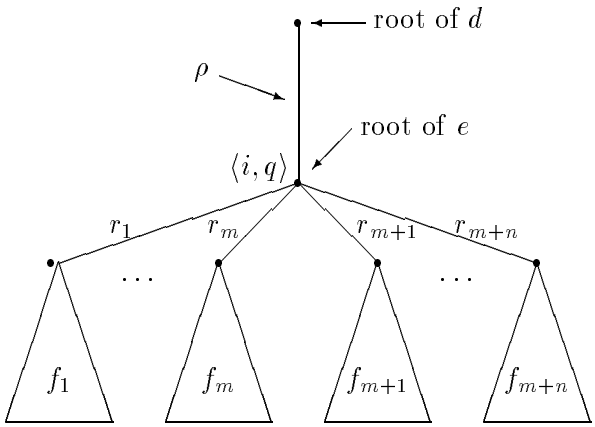
We are now ready to prove that every subtree e in $D^\tau(\rho)$ is representable by a term M. Without loss of generality, we assume that M has the form

$$\lambda^* x_1, \ldots, x_k\,.\,B$$

where B is an expression with free variables $x_1, \ldots, x_k$; for $k = 0$, this specializes to $M = B$. The construction of B proceeds by induction on the depth of subtrees. Thus, the remainder of the proof is a nested induction argument, where the induction hypothesis holds for all subtrees e' of *lower type* and all *smaller* subtrees e' of type τ—regardless of the context of e' in its parent tree.

Since $\mathbb{D}^\tau(\rho)$ is the union of two disjoint sets, $\mathbb{N}^{\mathbb{E}}_{\perp}$ and $\{\langle i, q, f\rangle \mid \ldots\}$, we must consider two cases: (i) $e \in \mathbb{N}^{\mathbb{E}}_{\perp}$ and (ii) $e = \langle i, q, f\rangle \in \mathbb{D}^\tau(\rho)$. In the first case, it is easy to verify that for any leaf $e \in \mathbb{N}$, the expression $M = \lambda^* x_1, \ldots, x_k\,.\,\ulcorner e \urcorner$ represents e. Similarly, if $e = \mathsf{error}_j$, then $M = \lambda^* x_1, \ldots, x_k\,.\,\mathsf{error}_j$ represents e. If $e = \perp$, then $M = \lambda^* x_1, \ldots, x_k\,.\,\Omega^\tau$ represents e. (Recall that Ω^τ abbreviates $\mathsf{Y}^\tau(\lambda^* x^\tau\,.\,x^\tau)$.)

In the second case, e is a finite subtree $\langle i, q, f\rangle$ compatible with the context ρ:



Let the proper domain of the branching function f be the union of the sets $\{r_1, \ldots, r_m\}$ and $\{r_{m+1}, \ldots, r_{m+n}\}$ where the responses is the first set have the form:

$$\begin{aligned} r_1 &= q[?/\langle i_1, p_1\rangle] \\ &\ldots \\ r_m &= q[?/\langle i_m, p_m\rangle] \end{aligned}$$

and the responses in second set have the form:

$$\begin{aligned} r_{m+1} &= q[?/a_1] \\ &\ldots \\ r_{m+n} &= q[?/a_n] \end{aligned}$$

where $a_1, \ldots, a_n \in \mathbb{N}$. We refer to the elements of the first set as *query* responses and to the elements of the second set as *final* responses. Each edge r_j points to a subtree $f_j \neq \bot$. By the induction hypothesis, each subtree f_j is representable by an expression F_j in SPCF.

We want to construct M so that

$$\mathsf{apply}(\mathcal{T}[\![M]\!], d_1, \ldots, d_k) = \mathsf{apply}(e, d_1, \ldots, d_k)$$

for all argument trees $d_1, \ldots, d_k$ in $\mathbb{D}^{\tau_1}, \ldots, \mathbb{D}^{\tau_k}$, respectively, such that $d_i \sqsupseteq \bigsqcup \rho(i)$. Since the root of e has the form $\langle i, q, f\rangle$, the code in the body of M must extract the node value at position q in d_i to determine which subexpression F_i to evaluate. We know that the argument d_i (bound to x_i in the body of M) has type τ_i of the form $\sigma_1 \to \ldots \sigma_l \to o$ where $(l \geq 0)$. Let $\widehat{q}$ denote the tree context determined by the query q (as defined in Definition 4.12). Naïvely, we might try to construct a body B that tries to compute the node value at position q in d_i by applying x_i to a list of argument expressions $A_1, \ldots, A_l$ representing the trees $\bigsqcup \widehat{q}(1), \ldots \bigsqcup \widehat{q}(l)$, respectively. In other words, M has the form

$$M = \lambda^* x_1, \ldots, x_k \,.\, E[(x_i\, A_1\ \ldots\ A_l)]\,,$$

for some evaluation context $E[\,]$ (see Definition 4.25). We know that we can construct each expression A_j by the induction hypothesis, because the type σ_j is smaller than type τ. If the subtree $d_i @ q$ is a leaf $a \in \mathbb{N}$, then the application

$$(x_i\, A_1\ \ldots\ A_l)$$

indeed returns the value a, which the evaluation context $E[]$ can use to select the appropriate subexpression F_j to evaluate. In this case, the naïve strategy works.

On the other hand, if the root of the subtree $d_i@q$ is a query asking more information about the jth argument to d_i, the naïve strategy fails because the argument expression A_j does not contain the specified information. The expressions $A_1, \ldots, A_l$ only encode the information embedded in q. To clarify this problem, let us consider an example. Assume that $\rho = \emptyset$, $e = \langle 1, ?, \langle\langle 1, ?\rangle, 17\rangle\rangle$, $d_1 = \langle 1, ?, \bot\rangle$, and $d_j = \bot$ for $j \geq 2$. Then

$$\mathsf{apply}(e, d_1, \ldots, d_k) = 17\,,$$

but the naïve strategy yields

$$M = \lambda^* x_1, \ldots, x_k \,.\, E[(x_1\, \Omega\ \ldots\ \Omega)]\,,$$

implying that

$$\mathsf{apply}(\mathcal{T}[\![M]\!], d_1, \ldots, d_k) = \bot\,.$$

To solve this problem, we need to embed code in the arguments $A_1, \ldots, A_l$ to perform non-local exits from the application $(x_i\, A_1\ \ldots\ A_l)$ back to the body B whenever d_i (the tree bound to x_i) asks for information about its arguments beyond what is specified in q. In SPCF, we can simulate a non-local exit by applying the catch operator to a λ-abstraction M' constructed from e. In particular, we can define a λ-abstraction

$$M' = \lambda^* y_1, \ldots, y_m \,.\, (x_i\, B_1 \ldots B_l)$$

that

- requests its jth argument when d_i dominates the *query* response r_j, and
- immediately returns the answer a_j when d_i dominates the *answer* response r_j.

In the first case, $(\mathsf{catch}\ M')$ returns the integer $j-1$, indicating that the code block M should execute the code block F_j. In the second case, $(\mathsf{catch}\ M')$ returns the integer $m + a_j$ indicating that M should execute the code block F_{m+j}.

We formalize these observations as follows. Let $[x_i := d_i]$ be the environment that binds x_i to d_i. The expression M' must satisfy two conditions:

Condition (i) If $d_i \sqsupseteq q[?/a_j]$, then $\mathcal{T}[\![M']\!]_{[x_i:=d_i]} = a_j$.

Condition (ii) If $d_i \sqsupseteq q[?/\langle h_j, p_j\rangle]$, then $\mathcal{T}[\![M']\!]_{[x_i:=d_i]} = \langle j, ?, g\rangle$ for some branching function g.

The key to constructing M' is defining the expressions B_h. Each expression B_h must evaluate the variable y_j if the node value at position q in d_i matches the response r_j. In addition, each expression B_h must satisfy the constraint $\widehat{q}(h) \sqsubseteq \mathcal{T}[\![B_h]\!]_{\mathcal{E}}$ for all environments $\mathcal{E}$ binding the variables $y_1, \ldots, y_m$ to values in $\mathbb{N}_\bot^{\mathbb{E}}$. The precise form of the expressions $B_1, \ldots, B_l$ depends on the query q. For the moment, let us assume that we can construct the argument expressions $B_1, \ldots, B_l$.

Given the expression M', M is easy to construct. The application $(\mathsf{catch}\ M')$ identifies the proper response r_j (if any) below the root of e corresponding to the node value at position q in d_i. More precisely, conditions (i) and (ii) on M' above imply that

$$\mathcal{T}[\![(\mathsf{catch}\ M')]\!]_{[x_i:=d_i]} = \begin{cases} a_j + m \in \mathbb{N} & \text{if } d_i \sqsupseteq q[?/a_j] \\ j - 1 \in \mathbb{N} & \text{if } d_i \sqsupseteq q[?/\langle i_j, p_j\rangle]\,. \end{cases}$$

where $[x_i := d_i]$ is the environment binding x_i to d_i. Given the index j corresponding to the response r_j from d_i, M can invoke the code F_j representing the subtree f_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,

$$\begin{aligned} M = \ & \lambda^* x_1^{\tau_1} \ldots x_k^{\tau_k}\,. \\ & \quad \mathbf{let}\ w = (\mathsf{catch}\ (\lambda^* y_1, \ldots, y_m\,.\,(x_i\ B_1 \ldots B_l)))\ \mathbf{in} \\ & \qquad (\mathsf{if0}\ w\ F_1 x_1 \ldots x_k \\ & \qquad \ldots \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}_\perp^{m-1}\ w)\ F_m x_1 \ldots x_k \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}_\perp^{a_1+m}\ w)\ F_{m+1} x_1 \ldots x_k \\ & \qquad \ldots \\ & \qquad (\mathsf{if0}\ (\mathsf{sub1}_\perp^{a_n+m}\ w)\ F_{m+n} x_1 \ldots x_k \\ & \qquad \Omega) \ldots)) \ldots) \end{aligned}$$

where

$$\mathsf{sub1}_\perp = \lambda x\,.\,(\mathsf{if0}\ x\ \perp\ (\mathsf{sub1} x))\,,$$

the expression $(\mathsf{sub1}_\perp^b\ N)$ abbreviates the b-fold application of $\mathsf{sub1}_\perp$ to the argument expression N, the expression $(\mathbf{let}\ w = P\ \mathbf{in}\ Q)$ abbreviates $((\lambda^* w\,.\,Q)\ P)$. The procedure $\mathsf{sub1}_\perp$ is identical to $\mathsf{sub1}$ except that it diverges on the input 0.

The proof that M represents e is a straightforward consequence of the properties of the expression $(\mathsf{catch}\ M')$ described above. There are two cases:

1. If $d_i \sqsupseteq q[?/a_j]$, then $\mathcal{T}[\![(\mathsf{catch}\ M')]\!]_{[x_i:=d_i]} = a_j + m$, implying that $M \equiv_{\mathcal{T}} F_{m+j}$ by the simplification process described in Figure 7.

2. If $d_i \sqsupseteq q[?/\langle i_j, p_j\rangle]$, then $\mathcal{T}[\![(\mathsf{catch}\ M')]\!]_{[x_i:=d_i]} = j - 1$, implying that $M \equiv_{\mathcal{T}} F_j$ by a simplification process similar to the one given in Figure 7.

To complete the proof of the lemma, we must

- construct the argument expressions $B_1, \ldots, B_l$, and
- prove that the expression M', constructed from $B_1, \ldots, B_l$, satisfies conditions (i) and (ii) identified above.

Let h be an index in the range $1, \ldots, l$. The construction of B_h falls into one of two cases. In the first case, no query response r_j asks for more information about the hth argument. In this case, we simply define B_h as the closed expression representing the tree $\bigsqcup \widehat{q}(i)$, which exists by the induction hypothesis because the type σ_i of B_h is smaller than τ.

$$
\begin{aligned}
M \equiv_{\mathcal{T}} \; & \lambda^* x_1^{\tau_1} \ldots x_k^{\tau_k} . \\
& (\mathsf{if0}\ \ulcorner a_j + m \urcorner\ (F_1\ x_1\ \ldots\ x_k) \\
& \ldots \\
& (\mathsf{if0}\ (\mathsf{sub1}_{\perp}^{m-1}\ \ulcorner a_j + m \urcorner)\ (F_m\ x_1\ \ldots\ x_k) \\
& (\mathsf{if0}\ (\mathsf{sub1}_{\perp}^{a_1+m}\ \ulcorner a_j + m \urcorner)\ (F_{m+1}\ x_1\ \ldots\ x_k) \\
& \ldots \\
& (\mathsf{if0}\ (\mathsf{sub1}_{\perp}^{a_n+m}\ \ulcorner a_j + m \urcorner)\ (F_{m+n}\ x_1\ \ldots\ x_k) \\
& \qquad \Omega) \ldots)) \ldots) \\
\equiv_{\mathcal{T}} \; & \lambda^* x_1^{\tau_1} \ldots x_k^{\tau_k} . \\
& (\mathsf{if0}\ \mathsf{false}\ (F_1\ x_1\ \ldots\ x_k) \\
& \ldots \\
& (\mathsf{if0}\ \mathsf{false}\ (F_m\ x_1\ \ldots\ x_k) \\
& (\mathsf{if0}\ \mathsf{false}\ (F_{m+1}\ x_1\ \ldots\ x_k) \\
& \ldots \\
& (\mathsf{if0}\ \mathsf{true}\ (F_{m+j}\ x_1\ \ldots\ x_k) \\
& \ldots \\
& (\mathsf{if0}\ \mathsf{false}\ (F_{m+n}\ x_1\ \ldots\ x_k) \\
& \qquad \Omega) \ldots)) \ldots) \\
\equiv_{\mathcal{T}} \; & \lambda^* x_1^{\tau_1} \ldots x_k^{\tau_k} . \\
& (F_{m+j}\ x_1\ \ldots\ x_k) \\
\equiv_{\mathcal{T}} \; & F_{m+j} .
\end{aligned}
$$

Figure 7: Simplification of $(\mathsf{catch}\ M')$ when $d_i \sqsupseteq q[?/a_j]$

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 $r_1 = q[?/\langle i_1, p_1\rangle]$, ..., $r_s = q[?/\langle i_s, p_s\rangle]$. Since x_i has type $\tau_i = \sigma_1 \to \ldots \to \sigma_l \to o$, we know that the type of B_h has the form

$$\sigma_h = v_1 \to \ldots v_t \to o$$

for some integer $t \geq 0$. We also know that the queries $p_1, \ldots, p_s$ extend the information in $\bigsqcup \widehat{q}(h)$. Consequently, we can form a tree

$$e_h = \bigsqcup \widehat{q}(h) \sqcup p_1[?/\langle t+1, ?, \perp\rangle] \sqcup \ldots \sqcup p_s[?/\langle t+s, ?, \perp\rangle]$$

with type

$$\sigma'_h = v_1 \to \ldots v_t \to \underbrace{o \to \ldots \to o}_{s-\text{times}} \to o$$

that grafts subtrees onto the tree $\widehat{q}(h)$. We can interpret $\bigsqcup \widehat{q}(h)$ as a tree of type σ'_h, so we can directly compare the behavior of the trees e_h and $\bigsqcup \widehat{q}(h)$ under application. If we apply the tree e_h to an argument list $c_1, \ldots, c_{t+s}$, it performs exactly the same computation as $\bigsqcup \widehat{q}(h)$ except when the information in $c_1, \ldots, c_{t+s}$ dominates $\widehat{p_j}$ for some $j, 1 \leq j \leq s$. In this case,

$$\mathsf{apply}(e_h, c_1, \ldots, c_{t+s}) = \mathsf{apply}(\langle t+j, ?, \perp\rangle, c_1, \ldots, c_{t+s}) = \begin{cases} \perp & \text{if } c_{t+j} \in \mathbb{N}_{\perp} \\ c_{t+j} & \text{if } c_{t+j} \in \{\mathsf{error}_1, \mathsf{error}_2\} \end{cases}$$

while

$$\mathsf{apply}(\bigsqcup \widehat{q}(h), c_1, \ldots, c_{t+s}) = \bot\,.$$

When $c_{t+j} \in \{\mathsf{error}_1, \mathsf{error}_2\}$, the trees e_h and $\bigsqcup \widehat{q}(h)$ produce different answers.

By the induction hypothesis, we can represent e_h by a closed expression E_h because its type is smaller than τ. Thus, we define the expression B_h as follows:

$$B_h = \lambda^* z_1, \ldots, z_t\,.(E_h\ z_1\ \ldots\ z_t\ y_{i_1}\ \ldots\ y_{i_s})\,.$$

The body of B_h evaluates parameter y_{i_j} of M' precisely when the node value at position q in d_i matches query response r_j below the root of e.

The meaning of B_h in an environment $\mathcal{E}$ depends on the values in $\mathbb{N}_\bot^{\mathsf{E}}$ that $\mathcal{E}$ assigns to the free variables $y_{i_1}, \ldots, y_{i_s}$. We claim that $\mathcal{T}[\![B_h]\!]_{\mathcal{E}}$ is the element

$$\begin{array}{rcl} b_h & = & \bigsqcup \widehat{q}(h) \sqcup \bigsqcup \{p_j[?/\bot] \mid \mathcal{E}(y_{i_j}) \in \mathbb{N}_\bot\} \sqcup \bigsqcup \{p_j[?/\mathcal{E}(y_{i_j})] \mid \mathcal{E}(y_{i_j}) \in \{\mathsf{error}_1, \mathsf{error}_2\}\} \\ & \sqsupseteq & \bigsqcup \widehat{q}(h)\,. \end{array}$$

To verify this claim, let $c_1, \ldots, c_t$ be elements of $\mathbb{D}^{v_h}, \ldots, \mathbb{D}^{v_t}$, respectively. It is easy to check that

$$\begin{array}{rcl} \mathsf{apply}(b_h, c_1, \ldots, c_t) & = & \mathsf{apply}(e_h, c_1, \ldots, c_t, \mathcal{E}(y_{i_1}), \ldots, \mathcal{E}(y_{i_s})) \\ & = & \mathsf{apply}(\mathcal{T}[\![E_h]\!], c_1, \ldots, c_t, \mathcal{E}(y_{i_1}), \ldots, \mathcal{E}(y_{i_s})) \\ & = & \mathsf{apply}(\mathcal{T}[\![B_h]\!]_{\mathcal{E}}, c_1, \ldots, c_t)\,. \end{array}$$

By extensionality, $\mathcal{T}[\![B_h]\!]_{\mathcal{E}} = b_h$.

Now we must show that the term M' satisfies conditions (i) and (ii). Let $\mathcal{E}$ be any environment that binds x_i to the tree d_i and binds $y_1, \ldots, y_m$ to elements in $\mathbb{N}_\bot^{\mathsf{E}}$:

Condition (i)**:** If $d_i \sqsupseteq q[?/a_j]$, then

$$\mathcal{T}[\![(x_i\ B_1\ \ldots\ B_l)]\!]_{\mathcal{E}} \sqsupseteq \mathsf{apply}(q[?/a_j], b_1, \ldots, b_l)\,.$$

Since $b_h \sqsupseteq \bigsqcup \widehat{q}(h)$ for all h, Corollary 4.15 implies that

$$\mathsf{apply}(q[?/a_j], b_1, \ldots, b_l) = \mathsf{apply}(a_j, b_1, \ldots, b_l) = a_j\,.$$

Since $\mathbb{N}_\bot^{\mathsf{E}}$ is flat,

$$\mathcal{T}[\![(x_i\ B_1\ \ldots\ B_l)]\!]_{\mathcal{E}} = a_j\,.$$

But the values of $y_1, \ldots, y_m$ in $\mathcal{E}$ were chosen arbitrarily. Hence, we can conclude that

$$\mathcal{T}[\![M']\!]_{[x_i := d_i]} = a_j\,.$$

Condition (ii)**:** If $d_i \sqsupseteq q[?/\langle i_j, p_j\rangle]$, then

$$\mathcal{T}[\![(x_i\ B_1\ \ldots\ B_l)]\!]_{\mathcal{E}} \sqsupseteq \mathsf{apply}(q[?/\langle i_j, p_j\rangle], b_1, \ldots, b_l)\,.$$

As in the first subcase, we conclude that

$$\mathsf{apply}(q[?/\langle i_j, p_j\rangle], b_1, \ldots, b_l) = \mathsf{apply}(\langle i_j, p_j\rangle, b_1, \ldots, b_l)\,.$$

By construction, $b_{i_j}@p_j \in \mathbb{E}$, but the specific value depends on $\mathcal{E}(y_j)$. Hence,

$$\mathsf{apply}(\langle i_j, p_j\rangle, b_1, \ldots, b_l) = \begin{cases} \bot & \mathcal{E}(y_j) \in \mathbb{N}_\bot \\ y_j & \mathcal{E}(y_j) \in \mathbb{E} \end{cases}$$

But notice that $\langle j, ?, \bot\rangle$ applied to $\mathcal{E}(y_1), \ldots, \mathcal{E}(y_m)$ has the same behavior as $\langle i_j, p_j\rangle$ applied to $b_1, \ldots, b_l$. Hence,

$$\begin{array}{rcl} \mathsf{apply}(\mathcal{T}[\![(x_i\ B_1\ \ldots\ B_l)]\!]_{\mathcal{E}}, \mathcal{E}(y_1), \ldots, \mathcal{E}(y_m)) & \sqsupseteq & \mathsf{apply}(\langle i_j, p_j\rangle, b_1, \ldots, b_l) \\ & = & \mathsf{apply}(\langle j, ?, \bot\rangle, \mathcal{E}(y_1), \ldots, \mathcal{E}(y_m)) \end{array}$$

By extensionality,

$$\mathcal{T}[\![(x_i\ B_1\ \ldots\ B_l)]\!]_{\mathcal{E}} \sqsupseteq \langle j, ?, \bot\rangle$$

implying that $\mathcal{T}[\![M']\!]_{[x_i:=d_i]} = \langle j, ?, g\rangle$ for some branching function g. This observation completes the proof of the lemma. ∎

6 Observable Sequentiality in SPCF

We define the meaning of the language SPCF using the tree model $\mathcal{T}$.

Definition 6.1. (*Semantic Definition of SPCF*) The semantic definition of SPCF is the meaning function $\mathcal{T}$ determined by the model $\mathcal{T}$ restricted to programs. ∎

Theorem 6.2 (Sequentiality of SPCF) *SPCF is 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 2.9: $(i)\ \mathcal{T}[\![C[M_1, \ldots, M_k]]\!] = n$; $(ii)\ \mathcal{T}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{T}[\![\Omega]\!]$. We must show that there is one argument position j that forces divergence. Since the model satisfies the equation β, we have:

$$\mathcal{T}[\![C[M_1, \ldots, M_k]]\!] = \mathcal{T}[\![\mathsf{apply}(\lambda^* x_1 \ldots x_k\,.\,C[x_1, \ldots, x_k]), M_1, \ldots, M_k)]\!].$$

Hence, it suffices to consider the denotations of the k-ary procedure

$$(\lambda^* x_1 \ldots x_k\,.\,C[x_1, \ldots, x_k])$$

The possible denotations of such a procedure are either elements of $\mathbb{N}_\bot^{\mathbb{E}}$ or triples of the form $\langle j, ?, f\rangle$ for some $j \leq k$ and some branching function f. Clearly, the first case contradicts the hypothesis of Definition 2.9. The second case specifies that the k-ary procedure probes its jth argument first. But this fact implies that the program $C[M_1, \ldots, M_k]$ diverges if the expression M_j in jth hole diverges—regardless of the expressions in the other holes. Hence, the program behaves sequentially. ∎

Indeed, SPCF satisfies a stronger condition than sequentiality: every procedure propagates any errors that are encountered during program evaluation. We formalize this stronger condition, called *error-sensitivity* as follows.

Definition 6.3. (*Error-Sensitivity*) Let L be a language based on λ-calculus. A semantic definition $\mathcal{P}$ for L is *error-sensitive* iff there exist two closed expressions E_1 and E_2, denoting distinct and inconsistent elements of a flat domain for the ground type, with the following property. Let $C[M_1, \ldots, M_k]$ be a terminating program where the phrases $M_1, \ldots, M_k$ are closed, $\mathcal{P}[\![C[M_1, \ldots, M_k]]\!] \in \mathbb{N}$, and $\mathcal{P}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{P}[\![\Omega]\!]$. Then there exists j such that

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

for all $M'_l, 1 \leq l \leq k$. ∎

It is easy to show that SPCF is error-sensitive.

Theorem 6.4 *SPCF is error-sensitive.*

Proof. By exactly the same analysis presented in the preceding proof of the sequentiality of SPCF, the program $C[\ldots]$ returns error_i if the jth argument is error_i, regardless of the values of the remaining arguments. ∎

Error-sensitivity implies sequentiality.

Theorem 6.5 *If a semantic definition $\mathcal{P}$ of a language L is error-sensitive, then it is sequential.*

Proof. Let $C[M_1, \ldots, M_k]$ be a terminating program such that $\mathcal{P}[\![C[M_1, \ldots, M_k]]\!] = n$ for some $n \in \mathbb{N}$ yet $\mathcal{P}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{P}[\![\Omega]\!]$. Since L is error-sensitive, there exists j such that

$$\mathcal{P}[\![C[M'_1, \ldots, M'_{j-1}, E_1, M'_{j+1}, \ldots, M'_k]]\!] = \mathcal{P}[\![E_1]\!]$$

and

$$\mathcal{P}[\![C[M'_1, \ldots, M'_{j-1}, E_2, M'_{j+1}, \ldots, M'_k]]\!] = \mathcal{P}[\![E_2]\!]$$

for all M'_i. By monotonicity,

$$\mathcal{P}[\![C[M'_1, \ldots, M'_{j-1}, \Omega, M'_{j+1}, \ldots, M'_k]]\!] \sqsubseteq \mathcal{P}[\![E_1]\!]$$

and

$$\mathcal{P}[\![C[M'_1, \ldots, M'_{j-1}, \Omega, M'_{j+1}, \ldots, M'_k]]\!] \sqsubseteq \mathcal{P}[\![E_2]\!].$$

Since E_1 and E_2 denote inconsistent ground values in a flat domain,

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

which is precisely what sequentiality demands. ∎

The error-sensitivity of SPCF implies that the tree model $\mathcal{T}$ is extensional. The domain of decision trees of type $\sigma \to \tau$ is isomorphic to the domain of error-sensitive continuous functions mapping $\mathbb{D}^\sigma$ into $\mathbb{D}^\tau$. But error-sensitivity does not make the decision tree model fully abstract. To represent all error-sensitive functions in SPCF, we need some mechanism within the language for determining the evaluation order of programs. We call this property of languages *observable sequentiality*.

Definition 6.6. (*Observable Sequentiality*) A semantic definition $\mathcal{P}$ of a language L based on the typed λ-calculus is *observably sequential* iff it is sequential and it satisfies the following property. Let C be a context and $M_1, \ldots, M_k$ be closed expressions and let $C[M_1, \ldots, M_k]$ be a program such that $\mathcal{P}[\![C[M_1, \ldots, M_k]]\!] \in \mathbb{N}$ yet $\mathcal{P}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{P}[\![\Omega]\!]$. Then there exists a program context $D[\ ]$ such that

$$\mathcal{P}[\![D[\lambda x_1 \ldots x_k \,.\, C[x_1, \ldots, x_k]]]\!] = \mathcal{P}[\![j]\!]$$

where j is the sequentiality index of C. ∎

The catch procedures make SPCF observably sequential.

Theorem 6.7 *SPCF is observably sequential.*

Proof. We have already shown that SPCF is error-sensitive. The remainder of the proof is trivial: simply set $D[\ ] = (\mathsf{add1}\,(\mathsf{catch}\ [\ ]))$. ∎

SPCF is not the first observably sequential language that has been studied in the context of the full abstraction problem. Berry and Currien [5] defined an observably sequential language called CDS0 in their work on sequential algorithms. They also constructed a fully abstract model for CDS0 based on sequential algorithms (which correspond to decision trees without errors), but the model is not extensional because the language is not error-sensitive.

Our experience with PCF suggests that error-sensitivity and observable sequentiality are necessary properties for the construction of extensional, fully abstract tree models for sequential languages. Error-sensitivity is required to obtain extensionality and observable sequentiality is required to obtain full abstraction. In the absence of error-sensitivity, procedures that are strict in several different arguments have multiple representations. Similarly, in the absence of observable sequentiality, higher order procedures cannot recognize the difference between procedures that are identical except for differences in the order in which they evaluate their arguments. In the next section, we identify some features in practical languages that interfere with the properties of error sensitivity and observable sequentiality.

7 Generalizing Observable Sequentiality

PCF is neither error-sensitive nor observably sequential. Consequently, the decision tree model for PCF is neither extensional nor fully abstract. However, the interesting issue is which *practical* programming languages with rich control operators are not error-sensitive or observably sequential. At this point we know of one prominent example: sequential languages with control delimiters [10], also referred to as 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 error-sensitive 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 % [22] where % is defined by the equation

$$[\![(\%\ e)]\!] = \begin{cases} [\![e]\!] & [\![e]\!] \in \mathbb{N}_\perp \\ i & [\![e]\!] = \mathsf{error}_i \in \mathbb{E} \end{cases} .$$

In PCF′, we can define an error-insensitive addition function:

$$+_{ei} = (\lambda xy\,.(+_l\ (\%\ x)\ (\%\ y))) = (\lambda xy\,.(+_r\ (\%\ x)\ (\%\ y))),$$

which demonstrates that PCF′ is not an error-sensitive language.

At this point, we do not know whether our construction solves the full abstraction problem for all sequential functional languages of practical interest. The preceding example shows that the problem remains open for practical languages that are sequential but not error-sensitive and observably sequential.

Acknowledgements. We thank Rama Kannegati and Dorai Sitaram for helping hone our intuitions about error-sensitivity and sequentiality. Discussions with Pierre-Louis Curien clarified the relationship of our work to sequential algorithms and convinced us that we were working in the tradition of the full abstraction literature; his comments on an early draft improved the presentation of the material. Steve Brookes pointed out a mistake in our original definition of sequentiality. John Gately found several errors in the proofs that we inserted in the expanded version of the paper.

Appendices

A K and S

A.1 The K Combinator

Lemma A.1 *For all $d \in \mathbb{T}^\sigma, e \in \mathbb{T}^\tau$,*

$$\mathsf{apply}(\mathsf{K}^{\sigma,\tau}, d, e) = d.$$

Proof. We will prove this lemma for finite trees and then use a continuity argument to extend the lemma to all trees. To prove the lemma for finite trees, we make the following claim about the subtrees of $\mathsf{K}^{\sigma,\tau}$.

Claim A.2 *Let $d \in \mathbb{D}^\sigma, e \in \mathbb{D}^\tau$ and let $q \in \mathbb{Q}^\sigma$ be a valid path in d ($d@q$ is defined). Then,* $\mathsf{apply}(K(q), d, e) = d@q$.

Proof (of Claim). The proof proceeds by induction on the size of the subtree $d@q$. Let

$$K'_q = \underline{\lambda} r\,.\begin{cases} a & r = q[?/a], a \in \mathbb{N} \\ \langle i+2, p, \underline{\lambda} r'.K(r\,.\langle r', ?\rangle)\rangle & r = q[?/\langle i, p\rangle] \end{cases} .$$

By the definition of the generating function K,

$$K(q) = \langle 1, q, K'_q\rangle\,.$$

A case analysis on the form of $d@q$ yields the following values for $\mathsf{apply}(K(q), d, e)$:

1. $d@q \in \mathbb{E}$: If $d@q = \bot$,

$$\mathsf{apply}(\langle 1, q, K'_q\rangle, d, e) = \bot = d@q.$$

   If $d@q = \mathsf{error}_1$ or $d@q = \mathsf{error}_2$ the same reasoning holds.

2. $d@q \in \mathbb{N}$: By the definition of K'_q,

$$\mathsf{apply}(\langle 1, q, K'_q\rangle, d, e) = \mathsf{apply}(d@q, d, e) = d@q.$$

3. $d@q = \langle i, p, f\rangle$: Again by the definition of K'_q:

$$\begin{array}{rcll}
\mathsf{apply}(\langle 1, q, K'_q\rangle, d, e) & = & \mathsf{apply}(K'_q(\langle i, p\rangle), d, e) & \\
& = & \mathsf{apply}(\langle i+2, p, \underline{\lambda} r' \,.\, K(q[?/\langle i, p\rangle] \,.\, \langle r', ?\rangle)\rangle, d, e) & \\
& = & \langle i, p, \underline{\lambda} r' \,.\, \mathsf{apply}(K(q[?/\langle i, p\rangle] \,.\, \langle r', ?\rangle), d, e)\rangle & \\
& = & \langle i, p, \underline{\lambda} r' \,.\, f(r')\rangle & \text{(by the induction hypothesis!)} \\
& = & \langle i, p, f\rangle & \\
& = & d@q\,. &
\end{array}$$

This case analysis exhausts all possible forms for $d@q$ proving the claim. ∎ (Claim)
From the claim and the definition of K, we can immediately conclude that for $d \in \mathbb{D}^\sigma$, $e \in \mathbb{D}^\tau$

$$\mathsf{apply}(\mathsf{K}^{\sigma,\tau}, d, e) = d.$$

The rest of the lemma follows from continuity. For arbitrary d and e,

$$\begin{array}{rcl}
\mathsf{apply}(\mathsf{K}, d, e) & = & \bigsqcup \{\mathsf{apply}(\mathsf{K}, d', e') \mid d' \sqsubseteq d, e' \sqsubseteq e, d', e' \text{ are finite } \} \\
& = & \bigsqcup \{d' \mid d' \sqsubseteq d, d' \text{ is finite } \} \\
& = & d\,,
\end{array}$$

which proves the lemma. ∎

A.2 The S Combinator

Before we can prove that $\mathsf{S}^{\sigma,\tau,\upsilon}$ satisfies its characteristic equation, we need to define the generating function S for S rigorously and prove that S is a well-formed decision tree. The generating function S is the least upper bound of an ascending chain of partial functions S_n mapping $\mathbb{Q}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon} \times \mathbb{Q}^{\sigma\to\tau\to\upsilon}$ into $\bigcup_\rho \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\rho)$. Since the generating functions S and T are mutually recursive, a truncated approximation function T_n for T must be defined in conjunction with S_n. The definition of S_n and T_n follows the same pattern of the of K_n in Section 4.3.

Definition A.3. (S, T, S_n, T_n) For $n \geq -1$, the functions S_n and T_n are defined by the recursion equations in Figure 8 where $n \in \mathbb{N}$. ∎

$$
\begin{array}{rcl}
S(p_S, q_1) & = & \bigsqcup_n S_n(p_S, q_1) \\
\\
S_{-1}(p_S, q_1) & = & \bot \\
S_0(p_S, q_1) & = & \bot \\
\\
S_{n+1}(p_S, q_1) & = & \\
\multicolumn{3}{l}{\quad \mathsf{let}\ \{d_2 = \widehat{p_S}(2);\ d_3 = \widehat{p_S}(3)\}} \\
\multicolumn{3}{l}{\quad \mathsf{in}} \\
\multicolumn{3}{l}{\quad \langle 1, q_1, \underline{\lambda} r_1 \in \mathcal{R}(q_1)\,.\, \left\{ \begin{array}{ll}
a & r_1 = q_1[?/a] \\
S_n(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], & r_1 = q_1[?/\langle 1, p\rangle], \\
\quad r_1\,.\,\langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle) & d_3@p \neq \bot \\
\langle 3, p, \underline{\lambda} s \in \mathcal{R}(p)\,. & r_1 = q_1[?/\langle 1, p\rangle], \\
\quad S_{n-1}(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle s, ?\rangle\rangle\rangle], & d_3@p = \bot \\
\qquad r_1\,.\,\langle s, ?\rangle)\rangle & \\
T_n(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], \mathit{encode}(d_2, p)) & r_1 = q_1[?/\langle 2, p\rangle] \\
\langle i+3, p, \underline{\lambda} s \in \mathcal{R}(p)\,. & r_1 = q_1[?/\langle i+2, p\rangle] \\
\quad S_n(p_S[?/\langle 1, q_1, \langle r_1, \langle i+3, p, \langle s, ?\rangle\rangle\rangle], & \\
\qquad r_1\,.\,\langle s, ?\rangle)\rangle &
\end{array} \right. \rangle} \\
T(p_S, q_2) & = & \bigsqcup_n T_n(p_S, q_2) \\
\\
T_{-1}(p_S, q_2) & = & \bot \\
T_0(p_S, q_2) & = & \bot \\
\\
T_{n+1}(p_S, q_2) & = & \\
\multicolumn{3}{l}{\quad \mathsf{let}^*\ \{r_1 = \widehat{p_S}(1);\ d_2 = \widehat{p_S}(2);\ d_3 = \widehat{p_S}(3);\ q_1[?/\langle 2, p\rangle] = r_1\}} \\
\multicolumn{3}{l}{\quad \mathsf{in}} \\
\multicolumn{3}{l}{\quad \langle 2, q_2, \underline{\lambda} r_2 \in \mathcal{R}(q_2)\,.\, \left\{ \begin{array}{ll}
S_n(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], & r_2 = q_2[?/a] \\
\quad r_1\,.\,\langle \mathit{shift}_1(r_2), ?\rangle) & \\
T_n(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], & r_2 = q_2[?/\langle 1, p\rangle], \\
\quad r_2\,.\,\langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle) & d_3@p \neq \bot \\
\langle 3, p, \underline{\lambda} s \in \mathcal{R}(p)\,. & r_2 = q_2[?/\langle 1, p\rangle], \\
\quad T_{n-1}(p_S[?/\langle 2, q_2, \langle r_2, \langle 3, p, \langle s, ?\rangle\rangle\rangle], & d_3@p = \bot \\
\qquad r_2\,.\,\langle s, ?\rangle)\rangle & \\
S_n(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], & r_2 = q_2[?/\langle i+1, p\rangle] \\
\quad r_1\,.\,\langle \mathit{shift}_1(r_2), ?\rangle) &
\end{array} \right. \rangle}
\end{array}
$$

Figure 8: Definition of the functions S_n, T_n for $(n \geq -1)$

Informally, the finite subtree $S_n(p_S, q_1)$ is simply the subtree at position p_S in S truncated at depth n. The arguments q_1 and q_2 for S_n and T_n, respectively, are redundant, but reconstructing their values from p_S requires a tedious case analysis.

To prove that S is well-defined, we need to introduce invariants Σ and $\mathbf{T}$ describing the domain of the functions S_n and T_n, respectively. Before we state the invariants, we need to introduce the idea of an *error variant*.

Definition A.4. (*Error variant*) Given a finite tree $d \in \mathbb{D}^\sigma$ that does not contain any leaves of the form error_1 or error_2, a finite tree $d' \in \mathbb{D}^\sigma$ is an error_i variant of d iff d' is identical to d except for one subtree position q where $d'@q = \mathsf{error}_i$ but $d@q \neq \bot$. ∎

Let $\Sigma(p_S, q_1)$ be the property that $p_S \in \mathbb{Q}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}$ and $q_1 \in \mathbb{Q}^{(\sigma\to\tau\to\upsilon)}$ satisfy the following conditions where $r_1 = \sqcup\, \widehat{p_S}(1)$, $d_2 = \sqcup\, \widehat{p_S}(2)$, and $d_3 = \sqcup\, \widehat{p_S}(3)$:

- $q_1 = r_1 \,.\, \langle r_p, ?\rangle$ where

$$\begin{cases} r_p = p[?/\mathsf{root}\,(d_3@p)] & r_1 = q[?/\langle 1, p\rangle] \\ r_p = p[?/\mathsf{root}\,(\mathit{apply}(d_2, d_3)@p)] & r_1 = q[?/\langle 2, p\rangle] \\ r_p \in \mathcal{R}(p) & r_1 = q[?/\langle i+2, p\rangle]\,. \end{cases}$$

- $\widehat{q_1}(i+2) = \widehat{p_S}(i+3)$ for all $i \geq 0$.
- $\sqcup\, \widehat{q_1}(2) = \mathsf{apply}(d_2, d_3)$.
- $\sqcup\, \widehat{q_1}(1) \sqsubseteq d_3$.
- For all error_i variants d_2' of d_2, there exists $p' \in \widehat{q_1}(2)$ such $\mathsf{apply}(d_2', d_3)@p' = \mathsf{error}_i$.
- For all error_i variants d_3' of d_3 such that $\sqcup\, \widehat{q_1}(1) \sqsubseteq d_3'$, there exists $p' \in \widehat{q_1}(2)$ such $\mathsf{apply}(d_2, d_3')@p' = \mathsf{error}_i$.
- For all $q_1[?/\langle 2, p\rangle] \in \mathcal{R}(q_1)$, there exists a query q such that $d_2@q = \bot$, $p = \mathit{shift}_1(q)$, and $\mathit{encode}(d_2, p) = q$.

In the invariant Σ, the query p_S identifies a position in the tree S where a node of the form $\langle 1, q_1, \Box\rangle$ appears. Note that r_1, d_2, d_3, and q_1 cannot contain any error elements because error elements cannot appear in queries (*e.g.* p_S) or responses (*e.g.* r_p).

We need an analogous invariant $\mathbf{T}$ for each position p_S where a node of the form $\langle 2, q_2, \Box\rangle$ appears in S. Let $\mathbf{T}(p_S, q_2)$ be the property that $p_S \in \mathbb{Q}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}$ and $q_2 \in \mathbb{Q}^{\sigma\to\tau}$ satisfy the following conditions where $r_1 = \sqcup\, \widehat{p_S}(1) = q_1[?/\langle 2, p\rangle]$, $d_2 = \sqcup\, \widehat{p_S}(2)$, and $d_3 = \sqcup\, \widehat{p_S}(3)$:

- $q_2 =$

$$\begin{cases} \mathit{encode}(d_2, p) & p_S = p_S'[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle],\ r_1 = q_1[?/\langle 2, p\rangle] \\ r_2 \,.\, \langle p[?/\mathsf{root}(d_3@p)], ?\rangle & p_S = p_S'[?/\langle 2, q_2', \langle r_2, ?\rangle\rangle],\ r_2 = q_2'[?/\langle 1, p\rangle] \\ r_2 \,.\, \langle s, ?\rangle & p_S = p_S'[?/\langle 3, p, \langle s, ?\rangle\rangle] \end{cases}$$

- $d_2@q_2 = \bot$.
- $\widehat{q_1}(i+2) = \widehat{p_S}(i+3)$ for all $i \geq 0$.
- $\sqcup\, \widehat{q_1}(2) = \mathsf{apply}(d_2, d_3)$.
- $\sqcup\, \widehat{q_1}(1) \sqsubseteq d_3$.
- $\mathsf{apply}(d_2, d_3)@p = \bot$.

- For all error_i variants d_2' of d_2, there exists $p' \in \widehat{r_1}(2)$ such that $\mathsf{apply}(d_2', d_3)@p' = \mathsf{error}_i$.
- For all error_i variants d_3' of d_3 such that $\sqcup\, \widehat{q_1}(1) \sqsubseteq d_3'$, there exists $p' \in \widehat{r_1}(2)$ such $\mathsf{apply}(d_2, d_3')@p' = \mathsf{error}_i$.
- For all $r_2' = q_2[?/a] \in \mathcal{R}(q_2)$, $a \in \mathbb{N}$, and $r_1 \,.\, \langle \mathit{shift}_1(r_2'), \langle 2, p' \rangle \rangle \in \mathcal{R}(r_1 \,.\, \langle \mathit{shift}_1(r_2'), ? \rangle)$, there exists a query q such that $(d_2 \sqcup r_2')@q = \bot$, $p' = \mathit{shift}_1(q)$, and $\mathit{encode}(d_2 \sqcup r_2', p') = q$.
- For all $r_2' = q_2[?/\langle j+1, p' \rangle] \in \mathcal{R}(q_1)$ and all p' such that

  $$r_1 \,.\, \langle \mathit{shift}_1(r_2'), \langle 2, p' \rangle \rangle \in \mathcal{R}(r_1 \,.\, \langle \mathit{shift}_1(r_2'), ? \rangle)\,,$$

  there exists a query q such that $(d_2 \sqcup r_2')@q = \bot$, $p' = \mathit{shift}_1(q)$, and

  $$\mathit{encode}(d_2 \sqcup r_2', p') = q\,.$$

We are finally ready to prove that S is well-formed.

Claim A.5 *The following two statements about S_n and T_n both hold:*

1. *For all p_S and q_1 satisfying the invariant Σ,*

   $$S_n(p_S, q_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S})\,.$$

2. *For all p_S and q_2 satisfying the invariant $\mathbf{T}$,*

   $$T_n(p_S, q_2) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S})\,.$$

Proof. The proof of the claim proceeds by induction on n. We assume that the claim holds for all $n' < n$ and prove that it holds for n.

1. $n \leq 0$: In the base case, $S_n(p_S, q_1) = \bot$ and $T_n(p_S, q_2) = \bot$, so the claim trivially holds.

2. $n > 0$: In the induction step, we must consider all queries p_S and q_1 in the call $S_n(p_S, q_1)$ satsifying the invariant Σ. Similarly, we must consider all queries p_S and q_2 in the call $T_n(p_S, q_2)$ satisfying the invariant $\mathbf{T}$.

   For $n > 0$, $S_n(p_S, q_1)$ has the form $\langle 1, q_1, g \rangle$ where the branching function g is defined at $r_1 \in \mathcal{R}(q_1)$ by cases. Since the invariant Σ holds, we know that $\widehat{p_S}(1) = \overline{q}_1$ and $r_1 \in \mathcal{R}(q_1)$, implying that the subtree $\langle 1, q_1, g \rangle$ is well-formed iff $g(r_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})$ for all $r_1 \in \mathcal{R}(q_1)$. Let's consider each of the cases in the definition of the branching function g:

   (a) $r_1 = q_1[?/a]$ and $g(r_1) = a \in \mathbb{N}$. We immediately conclude

   $$g(r_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})$$

   from the definition of $\mathbb{D}$.

(b) $r_1 = q_1[?/\langle 1, p\rangle]$, $d_3@p \neq \bot$, and

$$g(r_1) = S_{n-1}(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], r_1 \,.\, \langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle)\,.$$

By assumption, the response $r_1 \in \mathcal{R}(q_1)$, implying $\widehat{p_S} \cup \{(1, r_1)\}$ is a valid context in $\mathbb{C}$. Similarly, $s \in \mathcal{R}(p)$ because $\mathsf{root}\,(d_3@p) \sqsubseteq d_3@p$ and $d_3 \in \mathbb{D}^\sigma$. Consequently, the $r_1 \,.\, \langle s, ?\rangle$ is a legal query on argument 1 in context $\widehat{p_S} \cup \{(1, r_1)\}$. It is easy to confirm that the invariant Σ holds for the pair

$$(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle],\ r_1 \,.\, \langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle)\,.$$

As a result,

$$g(r_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})$$

by induction.

(c) $r_1 = q_1[?/\langle 1, p\rangle]$, $d_3@p = \bot$, and

$$g(r_1) = \langle 3, p, \underline{\lambda} s \in \mathcal{R}(p) \,.\, S_{n-2}(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_1 \,.\, \langle s, ?\rangle)\rangle\,.$$

As in the preceding case, $r_1 \in \mathcal{R}(q_1)$ and $\widehat{p_S} \cup \{(1, r_1)\}$ is a valid context in $\mathbb{C}$. In addition, $d_3@p = \bot$ implies that $p \in \mathcal{Q}((\widehat{p_S} \cup \{(1, r_1)\})(3))$ (since $d_3 = \sqcup\widehat{p_S}(3)$). Hence,

$$\langle 3, p, \bot\rangle \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})\,.$$

For each $s \in \mathcal{R}(p)$, we can confirm that the invariant Σ holds for the pair

$$(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], \ r_1 \,.\, \langle s, ?\rangle)\,,$$

implying

$$S_{n-2}(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_1 \,.\, \langle s, ?\rangle) \in D^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\} \cup \{(3, s)\})\,.$$

Hence, by induction,

$$g(r_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})\,.$$

(d) $r_1 = q_1[?/\langle 2, p\rangle]$ and

$$g(r_1) = T_{n-1}(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], \mathit{encode}(d_2, p))\,.$$

Since Σ holds for p_S, we can confirm that $\mathbf{T}$ holds for the pair

$$(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], \mathit{encode}(d_2, p))\,.$$

Hence, by induction,

$$g(r_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})\,.$$

(e) $r_1 = q_1[?/\langle i+2, p\rangle]$ and

$$g(r_1) = \langle i+3, p, \underline{\lambda} s{\in}\mathcal{R}(p)\,.\quad S_n(p_S[?/\langle 1, q_1, \langle r_1, \langle i+3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_1\,.\,\langle s, ?\rangle)\rangle$$

Since Σ holds for p_S and $r_1 = q_1[?/\langle i+2, p\rangle]$,

$$\langle i+3, p, \bot\rangle \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})\,.$$

In addition, for each $s \in \mathcal{R}(p)$, we can confirm that the invariant Σ holds for the pair

$$(p_S[?/\langle 1, q_1, \langle r_1, \langle i+3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_1\,.\,\langle s, ?\rangle)\,,$$

implying

$$S_n(p_S[?/\langle 1, q_1, \langle r_1, \langle i+3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_1\,.\,\langle s, ?\rangle) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\} \cup \{(i+3, s)\})\,.$$

Hence, by induction,

$$g(r_1) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(1, r_1)\})\,.$$

For $n > 0$, $T_n(p_S, q_2)$ has the form $\langle 2, q_2, g\rangle$ where the branching function g is defined by cases on responses $r_2 \in \mathcal{R}(q_2)$. Since the invariant $\mathbf{T}$ holds for (p_S, q_2), we know that $\widehat{p_S}(2) = d_2$ and $d_2@q_2 = \bot$, implying that the subtree $\langle 2, q_2, g\rangle$ is well-formed iff $g(r_2) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2)\})$ for all $r_2 \in \mathcal{R}(q_2)$. Let's consider each of the cases in the definition of the branching function g.

(a) $r_2 = q_2[?/a]$ and

$$g(r_2) = S_{n-1}(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], r_1\,.\,\langle \mathit{shift}_1(r_2), ?\rangle)$$

where $a \in N$. Since the invariant $\mathbf{T}$ holds for (p_S, q_2), we can confirm that the invariant Σ holds for

$$(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], r_1\,.\,\langle \mathit{shift}_1(r_2), ?\rangle)\,.$$

Hence, by induction

$$g(r_2) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2)\})\,.$$

(b) $r_2 = q_2[?/\langle 1, p\rangle]$, $d_3@p \neq \bot$, and

$$g(r_2) = T_{n-1}(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], r_2\,.\,\langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle)\,.$$

Since the invariant $\mathbf{T}$ holds for for (p_S, q_2), we can confirm that the invariant $\mathbf{T}$ holds for

$$(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], r_2\,.\,\langle p[?/\mathsf{root}\,(d_3@p)], ?\rangle)\,.$$

Hence, by induction

$$g(r_2) =\in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2))\}\,.$$

(c) $r_2 = q_2[?/\langle 1, p\rangle]$, $d_3@p = \bot$, and

$$g(r_2) = \langle 3, p, \underline{\lambda}s \in \mathcal{R}(p)\,.\,T_{n-2}(p_S[?/\langle 2, q_2, \langle r_2, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_2\,.\,\langle s, ?\rangle)\rangle\,.$$

Since the invariant **T** holds for (p_S, q_2), we can confirm that the invariant **T** holds for

$$(p_S[?/\langle 2, q_2, \langle r_2, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_2\,.\,\langle s, ?\rangle)\,.$$

Hence,

$$T_{n-2}(p_S[?/\langle 2, q_2, \langle r_2, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle]) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2)\} \cup \{(3, s)\})\,.$$

In addition, $d_3@p = \bot$ implies that $p \in \mathcal{Q}((\widehat{p_S} \cup (2, r_2))(3))$ (since $d_3 = \sqcup\widehat{p_S}(3)$). Hence,

$$\langle 3, p, \bot\rangle \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2)\})\,,$$

implying that

$$g(r_2) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2)\})\,.$$

(d) $r_2 = q_2[?/\langle i+1, p\rangle]$ and

$$g(r_2) = S_{n-1}(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], r_1\,.\,\langle \mathit{shift}_1(r_2), ?\rangle)\,.$$

Since the invariant **T** holds for for (p_S, q_2), we can confirm that the invariant Σ holds for

$$(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle], r_1\,.\,\langle \mathit{shift}_1(r_2), ?\rangle)\,.$$

Hence, by induction

$$g(r_2) = S_{n-1}(p_S[?/\langle 2, q_2, \langle r_2, ?\rangle\rangle]) \in \mathbb{D}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}(\widehat{p_S} \cup \{(2, r_2)\})\,.$$

This analysis of the possible form of r_2 concludes the proof of the claim. ∎(Claim)
Given the claim, it is easy to show that

$$\mathsf{S} \in \mathbb{T}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}\,.$$

The query ? clearly satisfies the invariant Σ. In addition, for all $n \geq 0$,

$$S_n(?, ?) \sqsubseteq S_{n+1}(?, ?)\,.$$

Hence,

$$\mathsf{S} = \bigsqcup_n \{S_n(?, ?)\} \in \mathbb{T}^{(\sigma\to\tau\to\upsilon)\to(\sigma\to\tau)\to\sigma\to\upsilon}\,.$$

Now we are finally ready to prove that S satisfies its characteristic equation.

Lemma A.6 *For all e_1, e_2, e_3 in appropriate domains,*

$$\mathsf{apply}(S(?, ?), e_1, e_2, e_3) = \mathsf{apply}(\mathsf{apply}(e_1, e_3), \mathsf{apply}(e_2, e_3))$$

Proof. As in Lemma A.1, we first prove the lemma holds for finite subtrees; the proof relies on the fact that the inputs e_1, e_2, and e_3 direct the application process from the root of a tree to the given subtree (see Lemma A.7 below). Given that, we can prove the full lemma via the continuity of apply. ∎

Lemma A.7 *Let p_S and q_1 be queries satisfying the invariant Σ and let e_1, e_2, e_3 be finite elements in appropriate domains such that $\overline{q_1} \sqsubseteq e_1$, $\bigsqcup \widehat{p_S}(2) \sqsubseteq e_2$, and $\bigsqcup \widehat{p_S}(3) \sqsubseteq e_3$. Then*

$$\mathsf{apply}(S(p_S, q_1), e_1, e_2, e_3) = \mathsf{apply}(\mathsf{apply}(e_1@q_1, e_3), \mathsf{apply}(e_2, e_3)).$$

Proof. The proof proceeds by induction on the form of $e_1' \stackrel{df}{=} e_1@q_1$:

1. $e_1' \in \mathbb{E}$: If $e_1' = \perp$, then

$$\begin{aligned} \mathsf{apply}(S(p_S, q_1), e_1, e_2, e_3) &= \perp \\ &= \mathsf{apply}(\perp, \mathsf{apply}(e_2, e_3)) \\ &= \mathsf{apply}(\mathsf{apply}(e_1', e_3), \mathsf{apply}(e_2, e_3)) \end{aligned}$$

   If $e_1' = \mathsf{error}_1$ or $e_1' = \mathsf{error}_2$ the same reasoning holds.

2. $e_1' \in \mathbb{N}$: Then,

$$\begin{aligned} \mathsf{apply}(S(p_S, q_1), e_1, e_2, e_3) &= \mathsf{apply}(e_1', e_1, e_2, e_3) \\ &= e_1' \\ &= \mathsf{apply}(e_1', \mathsf{apply}(e_2, e_3)) \\ &= \mathsf{apply}(\mathsf{apply}(e_1', e_3), \mathsf{apply}(e_2, e_3)) \end{aligned}$$

3. $e_1' = \langle 1, p, f_1 \rangle$: Then,

$$\mathsf{apply}(S(p_S, q_1), e_1, e_2, e_3) = \mathsf{apply}((\underline{\lambda} r_1 \in \mathcal{R}(q_1) \,.\, \cdots)(q_1[?/\langle 1, p\rangle]), e_1, e_2, e_3)\,.$$

   The argument e_1 is asking for the value of its first argument at position p. But e_1's first argument is e_3. The tree $S(p_S, q_1)$ does the translation. For the remainder of this proof, let $d_1 \stackrel{df}{=} \overline{q_1}$, $d_2 \stackrel{df}{=} \bigsqcup \widehat{p_S}(2)$, and $d_3 \stackrel{df}{=} \bigsqcup \widehat{p_S}(3)$. In addition, let r_3 be the response by e_3 to the query p. Hence, $r_3 \sqsubseteq e_3$ and $r_3 \in \mathcal{R}(p)$. We need to distinguish two subcases:

   (a) $d_3@p \neq \perp$: That is p_S already contains the response r_3 to querying e_3 at position p. Let r_1 denote $q_1[?/\langle 1, p\rangle]$. Then, by the definition of S,

$$\begin{aligned} \ldots &= \mathsf{apply}(S(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], r_1 \,.\, \langle r_3, ?\rangle), e_1, e_2, e_3) \\ &= \mathsf{apply}(\mathsf{apply}(f_1(r_3), e_3), \mathsf{apply}(e_2, e_3)) \qquad \text{(by induction hypothesis)} \\ &= \mathsf{apply}(\mathsf{apply}(\langle 1, p, f_1\rangle, e_3), \mathsf{apply}(e_2, e_3)) \qquad \text{(by definition of apply)} \\ &= \mathsf{apply}(\mathsf{apply}(e_1', e_3), \mathsf{apply}(e_2, e_3)) \end{aligned}$$

(b) $d_3@p = \bot$: Let r_1 denote $q_1[?/\langle 1, p\rangle]$. Then

$$\begin{array}{rcll}
\ldots & = & \mathsf{apply}(\langle 3, p, \underline{\lambda} s \in \mathcal{R}(p)\,.\,S(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle s, ?\rangle\rangle\rangle\rangle], r_1\,.\,\langle s, ?\rangle)\rangle, \\
 & & \quad e_1, e_2, e_3) & \\
 & = & \mathsf{apply}(S(p_S[?/\langle 1, q_1, \langle r_1, \langle 3, p, \langle r_3, ?\rangle\rangle\rangle\rangle], r_1\,.\,\langle r_3, ?\rangle), & \\
 & & \quad e_1, e_2, e_3) & \\
 & = & \mathsf{apply}(\mathsf{apply}(f_1(r_3), e_3), \mathsf{apply}(e_2, e_3)) & \text{(by induction hypothesis)} \\
 & = & \mathsf{apply}(\mathsf{apply}(\langle 1, p, f_1\rangle, e_3), \mathsf{apply}(e_2, e_3)) & \text{(by definition of } \mathsf{apply}) \\
 & = & \mathsf{apply}(\mathsf{apply}(e_1', e_3), \mathsf{apply}(e_2, e_3)) &
\end{array}$$

The cases for $(e_3@p) \in \mathbb{E}$ are simpler.

4. $e_1' = \langle 2, p, f_1\rangle$: Hence,

$$\mathsf{apply}(S(p_S, q_1), e_1, e_2, e_3) \;=\; \mathsf{apply}((\underline{\lambda} r_1 \in \mathcal{R}(q_1)\,.\,\cdots)(q_1[?/\langle 2, p\rangle]), e_1, e_2, e_3)\,.$$

Let $r_1 = q_1[?/\langle 2, p\rangle]$. Then,

$$\begin{array}{rcl}
\mathsf{apply}(S(p_S, q_1), e_1, e_2, e_3) & = & \mathsf{apply}(T(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], q_2), e_1, e_2, e_3) \\
 & = & \mathsf{apply}(\langle 2, q_2, \underline{\lambda} r_2 \in \mathcal{R}(q_2)\,.\,\cdots\rangle, e_1, e_2, e_3)
\end{array}$$

where $q_2 = \mathit{encode}(d_2, p)$ and $\mathit{shift}_1(q_2) = p$. Since q_2 is a legitimate query about e_2, $\overline{q_2} \sqsubseteq e_2$.

To complete the proof for $e_1' = \langle 2, p, f_1\rangle$, we invoke the following claim (labeled $\ddagger$), which we will prove later.

For any queries p_S' and q_2' satisfying the invariant $\mathbf{T}$

$$\mathsf{apply}(T(p_S', q_2'), e_1, e_2, e_3) = \left\{\begin{array}{ll}
w & w \in \mathbb{E} \\
\mathsf{apply}(S(p_S'[?/\langle 2, q_2', \langle q_2'[?/w], ?\rangle\rangle], & w \in \mathbb{N} \\
\quad d_1'\,.\,\langle \mathit{shift}_1(q_2'[?/w]), ?\rangle), e_1, e_2, e_3) & \\
\mathsf{apply}(S(p_S'[?/\langle 2, q_2', \langle q_2[?/\langle i+1, p''\rangle], ?\rangle\rangle], & w = \langle i, p'', \cdot\rangle \\
\quad d_1'\,.\,\langle \mathit{shift}_1(q_2'[?/\langle i+1, p''\rangle]), ?\rangle), e_1, e_2, e_3) &
\end{array}\right. \qquad (\ddagger)$$

where $e_2' \stackrel{df}{=} e_2@q_2'$ and $w \stackrel{df}{=} \mathsf{apply}(e_2', e_3)$.

The hypothesis of claim $\ddagger$ is clearly where $p_S' = p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle]$ and $q_2' = q_2$. In this case, we can rewrite the claim as:

$$\mathsf{apply}(T(p_S[?/\langle 1, q_1, \langle r_1, ?\rangle\rangle], q_2), e_1, e_2, e_3) = \left\{\begin{array}{ll}
w & w \in \mathbb{E} \\
\mathsf{apply}(\mathsf{apply}(f_1(p[?/w]), e_3), \mathsf{apply}(e_2, e_3)) & w \in \mathbb{N} \\
\mathsf{apply}(\mathsf{apply}(f_1(p[?/\langle i, p''\rangle]), e_3), \mathsf{apply}(e_2, e_3)) & w = \langle i, p'', \cdot\rangle
\end{array}\right.$$

where $e_2' \stackrel{df}{=} e_2@q_2$ and $w \stackrel{df}{=} \mathsf{apply}(e_2', e_3)$.

Given the claim ($\ddagger$), the final step in the proof for the case $e_1' = \langle 2, p, f_1\rangle$ is to verify that the right side of the preceding equation equals

$$\mathsf{apply}(\mathsf{apply}(e_1', e_3), \mathsf{apply}(e_2, e_3))\,.$$

By the definition of apply, we know that

$$\begin{aligned}
&\mathsf{apply}(\mathsf{apply}(e_1', e_3), \mathsf{apply}(e_2, e_3))\\
&\quad = \mathsf{apply}(\mathsf{apply}(\langle 2, p, f_1\rangle, e_3), \mathsf{apply}(e_2, e_3))\\
&\quad = \mathsf{apply}(\langle 1, p, \underline{\lambda} r_1 \in \mathcal{R}(p)\,.\,\mathsf{apply}(f_1(r_1), e_3)\rangle, \mathsf{apply}(e_2, e_3))\,,
\end{aligned}$$

independent of w. In addition,

$$\begin{aligned}
\mathsf{apply}(e_2, e_3) &\sqsupseteq \mathsf{apply}(q_2[?/e_2'], e_3)\\
&= \mathsf{apply}(\mathit{encode}(e_2, p)[?/e_2'], e_3)\\
&= p[?/\mathsf{apply}(e_2', e_3)]\\
&= p[?/w],
\end{aligned}$$

implying that

$$\mathsf{apply}(e_2, e_3)@p = w.$$

Given this fact, we can verify the compound equation by performing a case split on w:

(a) $w \in \mathbb{E}$: Since $\mathsf{apply}(e_2, e_3)@p = w$,

$$\mathsf{apply}(\langle 1, p, \underline{\lambda} r'\,.\,\mathsf{apply}(f_1(r'), e_3)\rangle, \mathsf{apply}(e_2, e_3)) = w$$

(b) $w \in \mathbb{N}$: Since $\mathsf{apply}(e_2, e_3)@p = w$,

$$\begin{gathered}
\mathsf{apply}(\langle 1, p, \underline{\lambda} r' \in \mathcal{R}(p)\,.\,\mathsf{apply}(f_1(r'), e_3)\rangle, \mathsf{apply}(e_2, e_3)) =\\
\mathsf{apply}(\mathsf{apply}(f_1(p[?/w]), e_3), \mathsf{apply}(e_2, e_3))
\end{gathered}$$

(c) $w = \langle i, p'', \cdot\rangle$: Since $\mathsf{apply}(e_2, e_3)@p = \langle i, p'', \cdot\rangle$,

$$\begin{gathered}
\mathsf{apply}(\langle 1, p, \underline{\lambda} r' \in \mathcal{R}(p)\,.\,\mathsf{apply}(f_1(r'), e_3)\rangle, \mathsf{apply}(e_2, e_3)) =\\
\mathsf{apply}(\mathsf{apply}(f_1(p[?/\langle i, p''\rangle]), e_3), \mathsf{apply}(e_2, e_3))
\end{gathered}$$

To complete the proof for $e_1' = \langle 2, p, f_1\rangle$, we still need to prove the claim ($\ddagger$). The proof proceeds by a nested induction on $e_2' \stackrel{df}{=} e_2@q_2'$. We perform a case analysis on the form of e_2' and $w \stackrel{df}{=} \mathit{apply}(e_2', e_3)$, which is summarized in the following table:

	$e_2' \in \mathbb{E}$	$e_2' \in \mathbb{N}$	$e_2' = \langle 1, p', \cdot\rangle$	$e_2' = \langle i+1, p'', \cdot\rangle$
$w \in \mathbb{E}$	**easy**	**empty**	**A**	**empty**
$w \in \mathbb{N}$	**empty**	**easy**	**A**	**empty**
$w = \langle i, p'', \cdot\rangle$	**empty**	**empty**	**A**	**B**

The cases marked **easy** are easy to prove, the cases marked **empty** are vacuously true, and the cases marked **A** and **B** need additional explanation:

A: Since $e_2' = \langle 1, p', \cdot \rangle$, we can immediately deduce that

$$\begin{aligned}&\mathsf{apply}(T(p_S', q_2'), e_1, e_2, e_3) = \\ &\quad \mathsf{apply}((\underline{\lambda} r_2 \in \mathcal{R}(q_2') \,.\, \cdots)(q_2'[?/\langle 1, p' \rangle]), e_1, e_2, e_3)\end{aligned}$$

If $\overline{p'} \sqsubseteq d_3' \stackrel{df}{=} \bigsqcup \widehat{p_S'}(3)$, then p_S' already contains the response s by e_3 to the query p'. Moreover, it is not bottom or an error value. Hence,

$$\ldots = \mathsf{apply}(T(p_S'[?/\langle 2, q_2', \langle q_2'[?/\langle 1, p' \rangle], ?\rangle\rangle], q_2'[?/\langle 1, p', \langle s, ?\rangle\rangle]), e_1, e_2, e_3) \,.$$

The rest follows from the inductive hypothesis because the T above is invoked with new values of p_S' and q_2' (as determined in the recursion equations defining T) that correspond to a smaller value of e_2'

On the other hand, if $d_3'@p' = \bot$, p_S' does not contain e_3's response to p'. Therefore, T generates the query p' to probe e_3:

$$\begin{aligned}&\mathsf{apply}(T(p_S', q_2'), e_1, e_2, e_3) = \\ &\quad \mathsf{apply}(\langle 3, p', \underline{\lambda} s \in \mathcal{R}(p') \,. \\ &\qquad\qquad T(p_S'[?/\langle 2, q_2', \langle q_2'[?/\langle 1, p' \rangle], \langle 3, p', \langle s, ?\rangle\rangle\rangle], q_2'[?/\langle 1, p', \langle s, ?\rangle\rangle])\rangle, \\ &\qquad e_1, e_2, e_3)\end{aligned}$$

Now, if $e_3@p' \in \mathbb{E}$, then $w \in \mathbb{E}$ and $\mathsf{apply}(T(p_S', q_2'), e_1, e_2, e_3) = w$. Otherwise, let $s = p'[?/e_3@p']$. Then,

$$\begin{aligned}&\ldots = \\ &\quad \mathsf{apply}(T(p_S'[?/\langle 2, q_2', \langle q_2'[?/\langle 1, p' \rangle], \langle 3, p', \langle s, ?\rangle\rangle\rangle], q_2'[?/\langle 1, p', \langle s, ?\rangle\rangle]), \\ &\qquad e_1, e_2, e_3)\end{aligned}$$

and the rest follows by the inductive hypothesis as above.

B: In this case, the claim follows immediately since

$$\begin{aligned}&\mathsf{apply}(T(p_S', q_2'), e_1, e_2, e_3) = \\ &\quad \mathsf{apply}(S(p_S'[?/\langle 2, q_2', \langle q_2'[?/\langle i+1, p' \rangle], ?\rangle\rangle], d_1' \,.\, \langle \mathit{shift}_1(q_2'[?/\langle i+1, p' \rangle]), ?\rangle), \\ &\qquad e_1, e_2, e_3)\end{aligned}$$

by the definition of T, where $d_1' \stackrel{df}{=} \bigsqcup \widehat{p_S'}(1)$.

The preceding case analysis completes the proof of claim ($\ddagger$).

5. $e_1' = \langle i+3, p, f_1 \rangle$ for $i \geq 1$: The proof of this case follows the same outline as the proof of Lemma A.1. ∎

B Equations for catch and error

Lemma B.1 *For all variables $x_1, \ldots, x_n$, for all evaluation contexts E, and for all $1 \leq j \leq n$,*

$$\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, E[x_j]]\!] = \langle j, ?, f \rangle$$

for an appropriate branching function f.

Proof. Let $d = \mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, E[x_j]]\!]$. We procced by induction on the structure of the evaluation context $E[\,]$. There are three possible forms for $E[\,]$

1. $E = [\ ]$: By (β), for all d_i in appropriate domains $\mathsf{apply}(d, d_1, \ldots, d_n) = d_j$. Thus, since d_j could be error_1, d cannot be $\bot$ or a constant. Similarly, d also cannot be error_1, because d_j could be $\bot$. Thus, d must be of the shape $\langle i, ?, f \rangle$ for some i and f. Assume $i \neq j$. Then, for $d_i = \bot$ and $d_j = \mathsf{error}_1$ we have that

   $$\mathsf{apply}(d, d_1, \ldots, d_n) = \mathsf{apply}(\langle i, ?, f \rangle, d_1, \ldots, d_n) = \bot \neq \mathsf{error}_1 = d_j,$$

   which means that we have again derived a contradiction. We conclude that

   $$\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, x_j]\!] = \langle j, ?, f \rangle$$

   for some function f.

2. $E = \mathsf{apply}(s, E')$: Observe that for all d_i in appropriate domains,

   $$\begin{array}{l}
   \mathsf{apply}(d, d_1, \ldots, d_n) \\
   \quad = \quad \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, \mathsf{apply}(s, E'[x_j])]\!], d_1, \ldots, d_n) \\
   \quad = \quad \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, \mathsf{apply}(s, \mathsf{apply}(\mathsf{apply}(\lambda^* x_1, \ldots, x_n \,.\, E'[x_j], \\
   \qquad\qquad x_1, \ldots, x_n)))]\!], \\
   \qquad d_1, \ldots, d_n) \\
   \quad = \quad \mathsf{apply}(\mathcal{T}[\![s]\!], \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, E'[x_j]]\!], d_1, \ldots, d_n)) \\
   \quad = \quad \mathsf{apply}(\langle 1, ?, f_s \rangle, \mathsf{apply}(\langle j, ?, f' \rangle, d_1, \ldots, d_n)) \,.
   \end{array}$$

   The last step is an immediate consequence of the inductive hypothesis for some function f' and the definition of strict, unary functions. As above, if $d_j = \mathsf{error}_1$, the right-hand side is error_1, and thus d cannot be $\bot$ or a constant. Conversely, if $d_j = \bot$ then the right-hand side is $\bot$, and therefore d cannot be error_1. If d is $\langle i, ?, f \rangle$, then $i = j$ since the negation leads to a contradiction as in the base case.

3. $E = \mathsf{apply}(E', M)$: With a short calculation we can see that the argument from the previous case applies here, too:

   $$\begin{array}{l}
   \mathsf{apply}(d, d_1, \ldots, d_n) \\
   \quad = \quad \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, \mathsf{apply}(E'[x_j], M)]\!], d_1, \ldots, d_n) \\
   \quad = \quad \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, \mathsf{apply}(\mathsf{apply}(\lambda^* x_1, \ldots, x_n \,.\, E'[x_j], x_1, \ldots, x_n), M)]\!],
   \end{array}$$

$$
\begin{aligned}
& d_1, \ldots, d_n) \\
= \;& \mathsf{apply}(\mathcal{T}[\![\mathsf{apply}(\lambda^* x_1, \ldots, x_n \,.\, E'[x_j], d_1, \ldots, d_n)]\!], \mathcal{T}[\![M[x_1/d_1] \ldots [x_n/d_n]]\!]) \\
= \;& \mathsf{apply}(\mathsf{apply}(\langle j, ?, f' \rangle, d_1, \ldots, d_n), \mathcal{T}[\![M[x_1/d_1] \ldots [x_n/d_n]]\!])
\end{aligned}
$$

where the last step follows from the inductive hypothesis for some function f'.

4. $E = \mathsf{apply}(\mathsf{catch}, \mathcal{T}[\![\lambda^* y_1, \ldots, y_m \,.\, E']\!])$: Finally, the case of catch applications is a straightforward modification of the second case:

$$
\begin{aligned}
& \mathsf{apply}(d, d_1, \ldots, d_n) \\
& \quad = \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, \mathsf{apply}(\mathsf{catch}, \lambda^* y_1, \ldots, y_m \,.\, E'[x_j])]\!], d_1, \ldots, d_n) \\
& \quad = \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n \,.\, \mathsf{apply}(\mathsf{catch}, \\
& \qquad\qquad \mathsf{apply}(\lambda^* x_1, \ldots, x_n, y_1, \ldots, y_m \,.\, E'[x_j], x_1, \ldots, x_n))]\!], \\
& \qquad d_1, \ldots, d_n) \\
& \quad = \mathsf{apply}(\mathsf{catch}, \mathsf{apply}(\mathcal{T}[\![\lambda^* x_1, \ldots, x_n, y_1, \ldots, y_m \,.\, E'[x_j]]\!], d_1, \ldots, d_n)) \\
& \quad = \mathsf{apply}(\langle 1, ?, f_{\mathsf{catch}} \rangle, \mathsf{apply}(\langle j, ?, f' \rangle, d_1, \ldots, d_n))
\end{aligned}
$$

where the final step follows from the inductive hypothesis for some function f'.

This analysis covers all the possible forms for $E[\,]$, completing the proof. ∎

References

1. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* Revised Edition. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984.
2. Berry, G. Séquentialité de l'evaluation formelle des λ-expressions. In *Proc. 3rd International Colloquium on Programming*, 1978.
3. Berry, G. *Modèles complètement adéquats et stables des lambda-calculus typé.* Ph.D. dissertation, Université Paris VII, 1979.
4. Berry, G. and P-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.* **20**, 1982, 265–321.
5. Berry, G. and P-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language cds. In *Algebraic Methods in Semantics*, edited by J. Reynolds and M.Nivat. Cambridge University Press. London, 1985, 35–88.
6. Berry, G., P-L. Curien, and P.-P. Lévy. Full-abstraction of sequential languages: the state of the art. In *Algebraic Methods in Semantics*, edited by J. Reynolds and M.Nivat. Cambridge University Press. London, 1985, 89–131.
7. Cartwright, R. and A. Demers. The topology of program termination. In *Proc. Symposium on Logic in Computer Science*, 1988, 296–308.

8. Curien, P-L. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, London. 1986. Birkhäuser, Revised Edition, to appear.
9. Curien, P.-L.. Observable algorithms on concrete data structures. In *Proc 7th Symposium on Logic in Computer Science*, 1992, to appear.
10. Felleisen, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 180–190.
11. 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.*, 1992, to appear.
12. 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.
13. 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, 1991, 131–151.
14. Kahn, G. and G. Plotkin. Structures des donnés concrètes (Domaines Concrètes). IRIA Report 336. 1978.
15. 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.
16. Milner, R. Fully abstract models of typed λ-calculi. *Theor. Comput. Sci.* **4**, 1977, 1–22.
17. Mulmuley, K. *Full Abstraction and Semantic Equivalences.* Ph.D. dissertation, Carnegie Mellon University, 1985. MIT Press, Cambridge, Massachusetts, 1986.
18. Plotkin, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.
19. Rose, J.R. and G.L. Steele Jr. C*: An extended C language for data parallel programming. In *Proc. Second International Conference on Supercomputing.* International Supercomputing Institute, Inc. (Santa Clara, 1987) Volume II, 2-16. Also available as: Technical Report No. PL87-5, Thinking Machines Corporation, 1987.
20. Scott, D. S. Domains for denotational semantics. In *Proc. International Conference on on Automata, Languages, and Programming*, Lecture Notes in Mathematics 140, Springer-Verlag, Berlin, 1982, ??–??.
21. Scott, D.S. *Lectures on a Mathematical Theory of Computation.* Techn. Monograph PRG-19, Oxford University Computing Laboratory, Programming Research Group, 1981.
22. 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.
23. Steele, G.L., Jr. *Common Lisp—The Language.* Digital Press, 1984.

24. Steele, G.L., Jr. and G.J. Sussman. The revised report on Scheme, a dialect of Lisp. Memo 452, MIT AI-Lab, 1978.
25. Stoughton, A. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Piman, London. 1986.
26. Vuillemin, J. Proof techniques for recursive programs. IRIA Report. 1973.