Observable Sequentiality and Full Abstraction

Robert Cartwright Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

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 lambda 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. We believe that the same construction will yield fully abstract models for functional languages with different control operators for observing the order of evaluation.

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 concrete 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 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 with traditional mathematical tools. 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 models based on the conventional continuous function spaces are typically not fully abstract for sequential languages. Consider the following family of procedures[1] p_i defined in a sequential, call-by-name functional language L:

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

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 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 the language L includes functions that evaluate their arguments in parallel. A simple example of a parallel func-

*Both authors were supported in part by NSF grant CCR 89-17022 and Darpa/NSF grant CCR 87-20277.

To appear in:
19th POPL
January 19–22, 1992
Albuquerque, NM

[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.

tion is *parallel-and*, which returns false if *either* input is false and returns true if *both* inputs are true. If we apply the conventional meaning of 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 [13, 20]. 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 [20]. Indeed, all commonly used deterministic languages are *sequential*. In informal terms, a language is sequential if it can be implemented without timeslicing 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* [17] are sequential.

1.1 Summary of Previous Work

Milner [14] and Plotkin [16] 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 above 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 because it does not identify any mathematical structure in program denotations other than their operational semantics. 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 later effort to understand the semantics of sequential languages, Berry and Curien [2, 3, 5, 8] constructed models for PCF with more restrictive domains of procedure denotations. Berry eliminated many parallel functions from the domain of procedure denotations 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 [3, 4] imposed further restrictions on the space of procedure denotations by interpreting procedures as *sequential algorithms over concrete domains* [12]. 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 procedure denotations 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 [15] 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 his construction relies on the syntactic notion of observational equivalence. For more details on the history of the full abstraction problem for sequential languages, we refer the reader to two extensive surveys [5, 23].

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 fully abstract denotational models 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 most *practical* languages, a programmer can observe the order in which a procedure evaluates its arguments by generating run-time errors. Similarly, many of these languages permit programs to determine and exploit the evaluation order of subexpressions by using explicit control operators such as exception 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

[2] This observation is due to A. Meyer (MIT): personal communication, August 1991

PCF in Section 2, we compare PCF to realistic functional languages in Section 3. Unlike practical versions of functional 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 by identifying error-sensitive functions with decision trees. We define such a model in Section 4 and show that it is extensional. In Section 5, we sketch a proof of the full abstraction theorem. Finally, in Section 6, we show that our extension of PCF is sequential and formulate the notions of error-sensitivity and observable sequentiality.

Due to space limitations, this paper only sketches the construction of the model and the proof of the full abstraction theorem. We refer the interested reader to an extended version of the paper [7].

2 PCF

PCF is a simple functional language based on the typed λ-calculus with numerals, increment and decrement procedures on numerals, conditionals, and a family of fixed-point combinators. The collection of types consists of a single ground type (o), denoting the set of non-negative integers, and procedure types ($\tau \to \sigma$), denoting sets of higher-order procedures, constructed from simpler types τ and σ. The set of types τ is formally defined by the recursive rule:

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

This rule is equivalent to

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

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.

The set of PCF terms ($M, M', \ldots$) consists of numerals ($\ulcorner n \urcorner$ for each $n \in \mathbb{N}$), procedural constants (f), typed variables (x_σ), abstractions, and term juxtapositions:

$$\begin{array}{rcl} M & ::= & \ulcorner n \urcorner \mid f \mid x_\sigma \mid (\lambda x_\sigma . M_\tau)_{\sigma\to\tau} \mid (M_{\sigma\to\tau} M_\sigma)_\tau \\ f & ::= & \mathsf{add1}_{o\to o} \mid \mathsf{sub1}_{o\to o} \mid \mathsf{if0}_{o\to o\to o\to o} \mid \mathsf{Y}_{(\sigma\to\sigma)\to\sigma} \end{array}$$

PCF *programs* are closed PCF terms of ground type. A *syntactic context* is a term with "holes" [ ] in place of some subexpressions. The term $C[M, \ldots, N]$ is the result of filling the holes of C with $M, \ldots, N$, possibly capturing free variables in $M, \ldots, N$ in the process.

In models of PCF, numerals denote numbers, λ-abstractions denote call-by-name procedures, juxtapositions denote procedure applications, and the procedural constants add1, sub1, if0, and Y have their usual meanings. The procedures add1 and sub1, respectively, add and subtract 1 from their integer inputs; the latter diverges on 0. The procedure 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 combinator computes the fixed points for procedures of appropriate type. In PCF program text, we also use the constant Ω, which abbreviates the term $\mathsf{Y}(\lambda x.x)$, the prototypical divergent expression. The following equations define *two* different versions of the binary addition procedure in PCF:

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

We will use these examples in subsequent discussions.

To define a denotational model for PCF or any functional language L based on the typed λ-calculus, we must define a family of domains $\mathbb{D}^\tau$ and a family of meaning functions $\mathcal{E}_\tau$. The domain $\mathbb{D}^\tau$ consists of the denotations for closed terms of type τ; $\mathcal{E}_\tau : L_\tau \longrightarrow \mathbb{D}^\tau$ is the meaning function that maps the set L_τ of closed terms of type τ to their meaning. Figure 1 contains the conventional function model of PCF; it interprets each type $\sigma \to \tau$ as the set of all *continuous* functions from type σ into type τ.

Given a denotational model $\mathcal{M}$ for the language L, we define the notion of observational equivalence and full abstraction as follows [14].

Definition 2.1. (*Denotational Equivalence, Observational Equivalence, Full Abstraction*) Two closed expressions, M and M', in L are *denotationally equivalent* in a model $\mathcal{M}$, written $M \equiv_{\mathcal{M}} M'$ iff $\mathcal{E}[\![M]\!] = \mathcal{E}[\![M']\!]$. They are *observationally equivalent*, written $M \simeq M'$, iff for all integer contexts $C[\ ]$ such that $C[M]$ and $C[M']$ are programs, $\mathcal{E}[\![C[M]]\!] = \mathcal{E}[\![C[M']]\!]$. A model $\mathcal{M}$ is *fully abstract* iff $M \simeq M'$ implies $M \equiv_{\mathcal{M}} M'$ for closed M, M'.

Remark. Observational equivalence only depends on the meanings of complete programs. Hence, it is independent of the choice of model as long as the models agree on the meanings of complete programs. ∎

The standard denotational model 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.

Given the denotational semantics determined by $\mathbb{D}^\tau$ and $\mathcal{E}_\tau$, we can also define what it means for the language L to be *sequential*.

Definition 2.2. (*Sequentiality*) L is *sequential* iff the following condition holds. Let M_1 through M_k be expressions and let $C[M_1, \ldots, M_k]$ be a program such that $\mathcal{E}[\![C[M_1, \ldots, M_k]]\!] = n$ for some $n \in \mathbb{N}$ yet $\mathcal{E}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{E}[\![\Omega]\!]$. Then there exists a j such that $\mathcal{E}[\![C[M'_1, \ldots, M'_{j-1}, \Omega, M'_{j+1}, \ldots, M'_k]]\!] = \mathcal{E}[\![\Omega]\!]$ for all M'_i. ∎

Domains:

$$\begin{array}{rcl} \mathbb{D}^{o} & = & \mathbb{N}_{\perp} \\ \mathbb{D}^{\tau\to\sigma} & = & [\mathbb{D}^{\tau} \longrightarrow_{c} \mathbb{D}^{\sigma}] \end{array}$$

Meaning Functions:

$$\mathcal{E}[\![e]\!] = [e]_{CL}$$

where

$$\begin{array}{rcl} [\ulcorner n \urcorner]_{CL} & = & \ulcorner n \urcorner \\ [f]_{CL} & = & f \\ [x]_{CL} & = & x \\ [(M_1\,M_2)]_{CL} & = & \mathsf{apply}([M_1]_{CL}, [M_2]_{CL}) \\ [\lambda x.M]_{CL} & = & \lambda^{*}x.[M]_{CL} \end{array} \qquad \begin{array}{rcl} \lambda^{*}x.x & = & \mathsf{I} \\ \lambda^{*}x.M & = & \mathsf{apply}(\mathsf{K}, M) \text{ if } x \text{ is not free in } M \\ \lambda^{*}x.\mathsf{apply}(M_1, M_2) & = & \mathsf{apply}(\mathsf{apply}(\mathsf{S}, \lambda^{*}x.M_1), \lambda^{*}x.M_2) \end{array}$$

and, for $l, m, n \in \mathbb{N}$,

$$\begin{array}{rcl} \mathsf{apply}(f,a) & = & f(a) \\ \mathsf{I}(x) & = & x \\ \mathsf{K}(x)(y) & = & x \\ \mathsf{S}(x)(y)(z) & = & [x(z)](y(z)) \end{array} \qquad \begin{array}{rcl} \mathsf{add1}(m) & = & m+1 \\ \mathsf{sub1}(m+1) & = & m \\ \mathsf{if0}(0)(m)(n) & = & m \\ \mathsf{if0}(l+1)(m)(n) & = & n \\ \mathsf{Y}(f) & = & \bigsqcup\{f^{i}(\perp) \mid i \in \mathbb{N}\} \end{array} \qquad \begin{array}{rcl} \mathsf{add1}(\perp) & = & \perp \\ \mathsf{sub1}(0) = \mathsf{sub1}(\perp) & = & \perp \\ \mathsf{if0}(\perp)(m)(n) & = & \perp \end{array}$$

FIGURE 1: The Continuous Function Space Model for PCF

It is straightforward to prove that PCF is sequential [16:Activity Lemma].

3 Observing Sequentiality

PCF omits many features that are essential in practical languages. One of the most glaring omissions is the lack of any provision for generating run-time errors and performing non-local exits. Although this design choice simplifies the definition of the language, it has thwarted efforts to construct fully abstract models for PCF and similar 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.

Using errors, programmers can observe the order of evaluation. The most obvious omission from PCF is a provision for computations to generate errors. Any practical implementation of PCF would signal an error for an application of the procedure $\mathsf{sub1}$ to $\ulcorner 0 \urcorner$ because the latter is an invalid input for the former. Although this behavior is inconsistent with the semantics of PCF, which requires $(\mathsf{sub1}\ \ulcorner 0 \urcorner)$ to diverge, it is far more informative from a programmer's point of view. It also 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*. 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 returns 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.

Using error handlers, 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 generating

errors [20]. For this reason, many practical languages include an error handling facility or a non-local control operator.

The original version of Scheme [22], for example, contained a lexically-scoped **catch** construct for implementing non-local exits from expressions. Hence, the evaluation of the expression

$$(\textbf{catch}\ e\ (\mathsf{add1}\ (e\ 0)))$$

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

$$G = (\lambda f.\textbf{catch}\ x\ (f\ (x\ 0)\ (x\ 1)))$$

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

To add this capability to PCF, we introduce a family of catch procedures with types $(\tau_1 \to \ldots \tau_n \to o) \to o$. If f is a procedure of type $(\tau_1 \to \ldots t_n \to o)$, then (catch f) returns either $i-1$ if f evaluates the ith argument first or $k+n$ if f is a constant procedure with result k. Alternatively, (catch 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 $j = 1, 2$. In PCF, the catch procedure has the same expressive power as Scheme's (downward) **catch** construct, but the catch procedure has a simpler semantic definition and 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 with constants error_1 and error_2 of type o and with catch procedures of type $(t_1 \to \ldots t_n \to o) \to o$ for all $t_1, \ldots, t_n$. ∎

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 [20]. To construct a fully abstract model for SPCF, we need to develop a new form of model that excludes parallel functions. The informal operational semantics of SPCF (particularly the catch operator) suggests that procedure denotations should have more internal structure than function graphs. In particular, they should explicate the order in which arguments are evaluated.

Consider the procedure

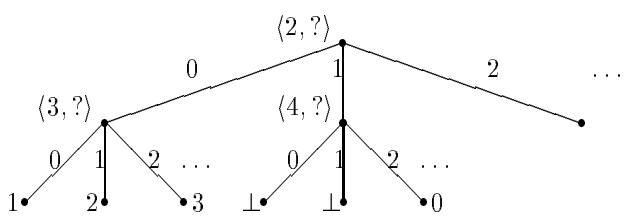
$$p = \lambda wxyz.\mathsf{if0}\ x\ (y+1)\ (z-2).$$

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 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 first argument is 0, p maps the third argument y to $y+1$; otherwise, it skips that argument and decreases the value of the fourth argument by 2:

$$\langle 2, \left\{ \begin{array}{l} \bot \mapsto \bot \\ \mathsf{error}_1 \mapsto \mathsf{error}_1 \\ \mathsf{error}_2 \mapsto \mathsf{error}_2 \\ 0 \mapsto \langle 3, \left\{ \begin{array}{l} \bot \mapsto \bot \\ \mathsf{error}_1 \mapsto \mathsf{error}_1 \\ \mathsf{error}_2 \mapsto \mathsf{error}_2 \\ 0 \mapsto 1 \\ 1 \mapsto 2 \\ \ldots \end{array} \right. \rangle \\ 1 \mapsto \langle 4, \left\{ \begin{array}{l} \bot \mapsto \bot \\ \mathsf{error}_1 \mapsto \mathsf{error}_1 \\ \mathsf{error}_2 \mapsto \mathsf{error}_2 \\ 0 \mapsto \bot \\ 1 \mapsto \bot \\ 2 \mapsto 0 \\ 3 \mapsto 1 \\ \ldots \end{array} \right. \rangle \\ \ldots \end{array} \right. \rangle$$

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

If we rotate this object, we can view it as a *decision tree*:

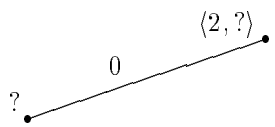


A node represents a query about the ith argument; for clarity, we will henceforth represent the argument index i by the pair $\langle i, ? \rangle$ in decision trees. The branches below a node are labeled with the possible values of the ith argument. The target node for each branch specifies what happens next. If it is a simple integer, $\bot$, or error, p has completed its computation; otherwise, it continues to query its argument(s). We have omitted the branches that map $\bot$ to $\bot$ and error_i to error_i to avoid cluttering decision trees with implied information.

Higher-order procedures explore trees sequentially. After deciding that procedures denote decision trees, it is natural for us to ask how higher-order procedures process procedural inputs. Given that trees resemble LISP S-expressions, we can visualize higher-order procedures as processes that examine decision trees in essentially the same manner as LISP procedures traverse S-expressions. To simplify notation, we express queries as patterns that match paths in decision trees. More specifically, a query is a rooted finite path of nodes and edges in a decision tree terminated by the marker "?". This marker identifies the node that the procedure wants to inspect. The *answer* produced by the query is the information stored in the specified node. If this match yields $\bot$ or error_i, the result of the entire process is $\bot$ or error_i. Like a LISP procedure, a higher-order procedure can inspect a node in an input tree only after inspecting its predecessors.

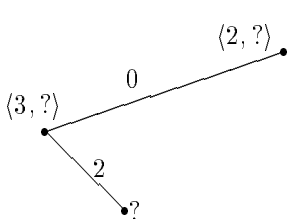
To illustrate this process, let us examine the behavior of the procedure $F = \lambda f.(\mathsf{add1}\,(f\,\Omega\,0\,2\,\mathsf{error}_1))$. When (the tree representing) F is applied to an argument tree f, it knows nothing about the value of f. Since F needs to know how f behaves on certain arguments, it asks the question "$\langle 1, ?\rangle$" to determine the root of f. If the value of f is the procedure p described above, the response to this question is $\langle 2, ?\rangle$, which is the contents of the root of p.

Now F knows that f first demands information about its second argument. To find out more about f, F must provide the information that f has requested, which is 0. This information determines the subtree within f that F wants to inspect. The information is specified by extending the previous response by the edge:



Matching this pattern against p yields the node value $\langle 3, ?\rangle$, which is a query about p's third argument.

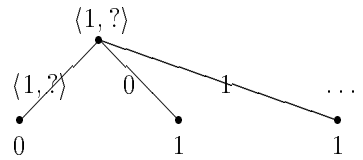
F passes the value 2 as the third argument to f. Consequently, F's next query to f has the form:



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

The preceding query-response protocol can be represented by a tree whose nodes are labeled with queries and whose arcs are labeled with the possible answers. Figure 2 shows part of the denotation of F, including the path that we just discussed. It also includes paths that 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 denoting 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 unary constant procedures to 1, and strict 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* continous functions. The presence of error values in the ground domain combined with the convention that all procedures are *error-sensitive* makes evaluation information *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 present within error-sensitive functions.[3]

4 The Tree Model

A model of SPCF has essentially the same form as a model of PCF. It consists of a family of Scott domains $\mathbb{D}^\tau$ and a family of meaning functions $\mathcal{E}_\tau$. In our model of SPCF, the former are domains consisting of the decision trees described in the previous section. Each meaning function $\mathcal{E}_\tau$ is a map from closed SPCF terms to trees in the appropriate domain. To define these meaning functions, we rely on extensions of the translation and abstraction algorithms, $[\cdot]_{CL}$ and λ^* for PCF (see Figure 1). However, we interpret the combinators produced by this translation as decision trees; the function

[3] Berry and Curien [3] 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 $+_r$ denote distinct concrete sequential algorithms but denote the same function [3:316]. In this framework, evaluation information is *intensional*. In later work [4, 8], Berry and Curien noted that sequential algorithms can distinguish different algorithms for the same function[8:210]. But they did not make a connection between intensional procedures and control operators like **catch**.

Recent discussions with Curien have revealed a surprising relationship between Berry and Curien's model for PCF and our model for SPCF. If we restrict our model to the operators in PCF and project error values onto $\bot$, our model is isomorphic to theirs [P.L. Curien (ENS): personal communication, May 1991].

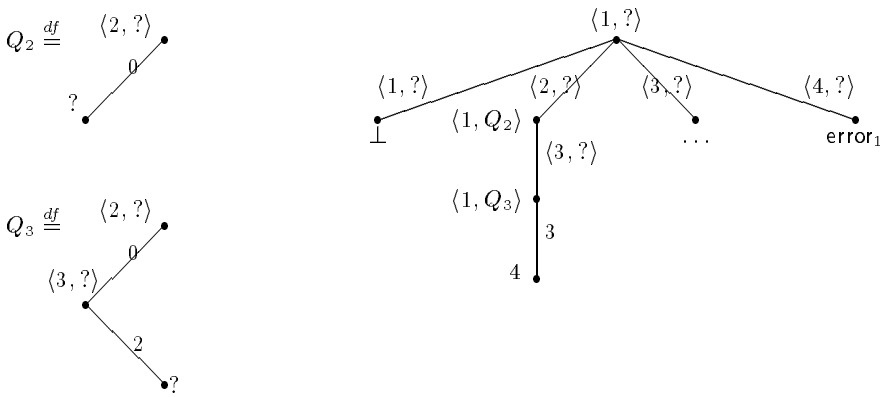


FIGURE 2: A Fragment of Procedure F

$\mathsf{apply} : \mathbb{D}^{\sigma\to\tau} \times \mathbb{D}^{\sigma} \longrightarrow \mathbb{D}^{\tau}$ converts decision trees into functions.

In the following subsection, we define our tree domains. Next, we define the apply functions and prove that our tree representation is extensional: procedural trees are identical precisely when they cannot be distinguished by apply. In the third and fourth subsection, we assign interpretations to the procedural constants and the combinators associated with SPCF, and prove the basic equational properties of the tree model.

4.1 Domains

A Scott domain is the ideal completion of a *finitary* basis [6, 18, 19]. A finitary basis B is a countable, partially ordered set such that every finite, bounded subset has a least upper bound; the principal ideals corresponding to elements 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 consistently complete, ω-algebraic cpo. In discussing domains, it is convenient to ignore the distinction between elements in 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 lower-type arguments. For each domain $\mathbb{D}^{\tau}$, we define a set of queries $\mathbb{Q}^{\tau}$, which higher-type objects can ask about elements of type τ, and a set of proper responses $\mathbb{R}^{\tau}$ to such queries. Since the behavior of error-sensitive functions on the answers $\{\bot, \mathsf{error}_1, \mathsf{error}_2\}$ is predetermined, we do not include these answers within the set of *proper responses* that appear in tree representations. The precise definition of $\mathbb{D}^{\tau\to\sigma}$ is rather subtle. We start by giving a preliminary definition that contains too many elements. Then we refine that definition to achieve a one-to-one correspondence between decision trees and observably sequential functions.

The definition of the set of domains $\mathbb{D}^{\tau}$ for each type τ proceeds by induction on the structure of τ. For procedure domains, we use the expansion of the type into a map from arbitrary k-tuples to ground types so that we can refer to all possible argument indices.

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

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

where

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

The approximation ordering 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 proper responses to this query are integers. Consequently, we define the set of ground queries and proper responses as follows:

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

Type $\tau = \tau_1 \to \ldots \to \tau_k \to o$: Finite higher-order objects are finite decision trees. Every path within such a tree is finite and all leaves are elements of $\mathbb{N}^{e}_{\bot}$. In addition, only finitely many branches below each non-terminal node are not $\bot$. Thus, we can inductively define the set of finite elements of $\mathbb{D}^{\tau}$ follows. A finite tree of type τ is either:

1. a leaf in $\mathbb{N}^{e}_{\bot}$, or
2. a triple $\langle i, q, f\rangle$ consisting of an index i, $1 \le i \le k$, a query q of type τ_i about the ith argument, and *branching* function f that maps proper responses to finite trees. For all but a finite set of proper responses, called the *proper domain* of f, the function f must produce the value $\bot$.

In pictures, we represent a tree $\langle i, q, f\rangle$ as a node labeled $\langle i, q\rangle$ with an outgoing edge labeled r for each

Elements :

$$\mathbb{D}^\tau(\rho) = \mathbb{N}_\perp^e \cup \{\langle i,q,f\rangle \,|\, 1{\le}i{\le}k, q{\in}\mathcal{Q}^{\tau_i}(\rho(i)),\ f : r \in \mathcal{R}^{\tau_i}(q) \mapsto d \in \mathbb{D}^\tau(\rho \cup \{\langle i,r\rangle\}),\ \{r \,|\, f(r) \neq \perp\} \text{ is finite}\}$$

Paths:

$$\mathbb{P}^\tau(B,\rho) = B(\rho) \cup \{\langle i,q,\langle r,p\rangle\rangle \,|\, 1 \le i \le k, q \in \mathcal{Q}^{\tau_i}(\rho(i)), r \in \mathcal{R}^{\tau_i}(q), p \in \mathbb{P}^\tau(B, \rho \cup \{\langle i,r\rangle\})\}$$

Queries :

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

Legal Queries :

$$\mathcal{Q}^\tau(\rho) = \begin{cases} \{?\} & \text{if } \rho = \emptyset \\ \{q \in \mathcal{Q}'^\tau(r) \,|\, r \in \rho, \text{ for all } r \in \rho : \overline{q} \not\sqsubseteq r\} & \text{if } \rho \neq \emptyset \end{cases}$$

$$\begin{aligned} \mathcal{Q}'^\tau(n) &= \emptyset \\ \mathcal{Q}'^\tau(\langle i,q\rangle) &= \{\langle i,q,\langle r,?\rangle\rangle \,|\, r \in \mathcal{R}^{\tau_i}(q)\} \\ \mathcal{Q}'^\tau(\langle i,q,\langle r,r'\rangle\rangle) &= \{\langle i,q,\langle r,q'\rangle\rangle \,|\, q' \in \mathcal{Q}'^{\tau_i}(r')\} \end{aligned}$$

Responses :

$$\mathbb{R}^\tau = \mathbb{P}(\underline{\lambda}\rho.\mathbb{N} \cup \{\langle i,q,\perp\rangle \,|\, i{\le}k, q \in \mathcal{Q}^{\tau_i}(\rho(i))\},\emptyset)$$

Legal Responses :

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

FIGURE 3: Elements, queries, and responses at $\tau = \tau_1 \to \ldots \to \tau_k \to o$

response in the proper domain of f; each edge r is attached to the subtree $f(r)$.

For each type τ, we tentatively define the set of queries $\mathbb{Q}^\tau$ and responses $\mathbb{R}^\tau$ as follows. Queries and responses of type τ designate paths in trees of type τ; they have exactly the same concrete representation as finite trees with two exceptions. First, the domains of branching functions in queries and responses contain at most one element. Second, the degenerate query is denoted by the special marker "?" and the final branching function in a query must map a response r to "?". Within a path, we denote the branching function that maps the response r to path p by the pair $\langle r,p\rangle$. Similarly, we denote the everywhere undefined branching function (which may occur at the end of a response) by $\perp$. The "?" marker at the end of each query identifies the node value that the query is seeking. Replacing the "?" marker in a query q with a final answer in $\mathbb{N}$ or an intermediate answer $\langle i,q,\perp\rangle$ yields a response to query q. The notation $q[?/N]$ denotes the response to q obtained by replacing "?" with the tree node N. 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 $\perp$; we define $\overline{q}$ as $q[?/\perp]$.

For the sake of brevity in notation, we abbreviate the triple $\langle i,q,\perp\rangle$ by the pair $\langle i,q\rangle$. Similarly, in pictures, we represent the response labeling each edge by the answer that replaces "?" in the corresponding query instead of the entire response.

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 node N in an argument tree before inspecting N. Finally, the definition of $\mathbb{R}^\tau$ does not guarantee that the responses to a query are appropriate.

We can solve these three problems by maintaining an environment that accumulates the responses that appear on a path from the root of a procedure tree to a given node. We refer to this environment as the *tree context* of a node. We will use the term *context* in place of tree context, unless it is ambiguous. Technically, a context is a relation mapping argument indices to responses; we use it as if it were a set-valued function. A context contains the knowledge that the procedure has gathered about its arguments at a given point in a computation. More precisely, given the context ρ of a node N in a procedure P, $\bigsqcup \rho(i) \in \mathbb{D}^{\tau_i}$ is the finite approximation for argument i of type τ known to P at node N during a computation.

Given the context of a node N, we can determine the set of queries about each argument that are *legal* at

node N: a query q with index i must extend the approximation tree $\bigsqcup \rho(i)$ for the ith argument 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 N in an argument tree are inspected before N is inspected.

We 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; the definition is given in Figure 3. In this definition, the auxiliary function $\mathcal{Q}'$ maps a legal response r to a set of queries extending the response by a branch $\langle r', ? \rangle$ where a is a legal answer to the query at the end of r. This set is empty if the response terminates with a final answer rather than a query. Pictorially, we have:

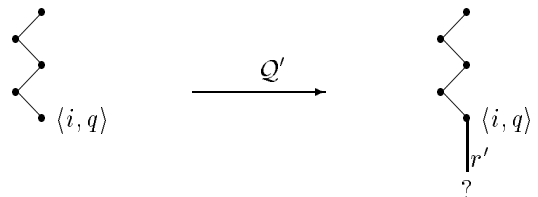


Figure 3 contains the formal definition of $\dot{\mathcal{Q}}'$.

The last problem that we must address is the definition of the legal responses to a query q. But this question is easy: a response $q[?/N]$ to q is legal iff the node N can appear at the point specified by q in the selected argument tree. This test reduces to confirming that the response $q[?/N]$ is a well-formed tree. Since arguments are trees of lower type, this reduction is not circular.

Based on the preceding analysis, we can now finalize the definition of finite elements for procedure types. We parameterize the definition of the sets $\mathbb{D}^\tau$ over contexts ρ and demand that subtrees of the shape $\langle i, q, f \rangle \in \mathbb{D}^\tau$ satisfy the following conditions:

1. $q \in \mathcal{Q}(\rho(i))$,
2. $f : r \in \mathcal{R}(q) \mapsto d \in \mathbb{D}^\tau(\rho \cup \{\langle i, r \rangle\})$, and
3. $\{r \mid f(r) \neq \bot\}$ is finite.

The inductive definition is summarized in a set of equations in Figure 3. The set $\mathbb{D}(\rho)$ contains all subtrees appropriate for tree context ρ. The domain of finite elements is made up of the subtrees that exits in the empty context.

The approximation order on this domain is the expected one. If d_1, d_2 are trees in $\mathbb{D}^\tau$ then $d_1 \sqsubseteq d_2$ if d_1 is a prefix of d_2.

Definition 4.1. (*Approximation*) Let ρ be a context, and let d, d' be subtrees in $\mathbb{D}(\rho)$. Then, $d \sqsubseteq d'$

1. if $d = \bot$; or
2. if $d = \langle i, q, f \rangle, d = \langle i, q, g \rangle$ and $f(r) \sqsubseteq g(r)$ for all $r \in \mathcal{R}(q)$.

∎

This finishes the formal definition of the finite elements of the family of domains $\mathbb{D}^\tau$. It is tedious but straightforward to prove that the functions $\mathcal{Q}^\tau$ and $\mathcal{R}^\tau$ are well-defined and that $\mathbb{D}^\tau$ is a finitary basis. The ideal completion process adds *infinite* trees to the finite elements in each domain $\mathbb{D}^\tau$.

4.2 Application

The function $\mathsf{apply} : \mathbb{D}^{\sigma \to \tau} \times \mathbb{D}^\sigma \longrightarrow \mathbb{D}^\tau$ interprets decision trees as functions. Given a subtree of a procedural tree and an (entire) argument tree, it produces a subtree for the appropriate context. More specifically, for inputs f in $\mathbb{D}^{\sigma \to \tau}$ and x in $\mathbb{D}^\sigma$, $\mathsf{apply}(f, x)$ 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 is a tree of the form $\langle 1, q, f' \rangle$, apply performs a pattern match between the specified query and the argument x. For proper responses r, apply selects the appropriate branch in the branching function, $f'(r)$, and continues the process of constructing the output tree. The improper answers error_1, error_2, and $\bot$ are mapped to error_1, error_2, and $\bot$, respectively.
3. If f is a tree of the form $\langle i+1, q, f' \rangle$, apply constructs a new decision tree of type τ of the form $\langle i, q, g' \rangle$ where g' is constructed by recursively processing each subtree $f'(r)$ for each response r.

Figure 4 contains the formal definition of the function apply including the definition of the auxiliary function *match*. The *match* function compares a query q with an element x and returns a response replacing "?" in q with contents of the node that matches "?" in x. If the matched node is an error value or bottom, *match* returns the node value instead of a response.

It is easy to show that apply and *match* are well-defined, *i.e.*, apply only uses *match* on arguments for which it is defined, and apply returns a subtree of the stipulated type and tree context. The function apply determines an *approximable mapping* on finite decision trees [19]. Consequently, we can extend apply to a continuous function on arbitrary elements. More importantly, we can prove that the model is *order-extensional*.[4]

[4] 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 $\langle 1, ?, \lambda x.\mathsf{error} \rangle$ are incomparable, yet for all d, $\mathsf{apply}(\langle 1, ?, \lambda x.\mathsf{error} \rangle, d) \sqsubseteq \mathsf{apply}(\mathsf{error}, d)$. On the other hand, while a model based on a single error value is not order-extensional, it is still extensional. We also conjecture that such a model is fully abstract. — S. Brookes (CMU) [personal communication, October 1991] suggested a modified approximation

$$
\begin{array}{rcl}
\mathsf{apply}(d, d_1) & = & d \qquad \text{if } d \in \mathbb{N}^e_\perp \\
\mathsf{apply}(\langle 1, q, f\rangle, d_1) & = & \begin{cases} r & r \in \mathbb{E} \\ \mathsf{apply}(f(r), d_1) & r \in \mathbb{R} \end{cases} \\
 & & \text{where } r = match(q, d_1) \\
\mathsf{apply}(\langle i+1, q, f\rangle, d_1) & = & \langle i, q, \underline{\lambda} r_i.\mathsf{apply}(f(r_i), d_1)\rangle
\end{array}
$$

$$
\begin{array}{rcl}
match(?, d) & = & d \qquad \text{if } d \in \mathbb{N}^e_\perp \\
match(?, \langle i, q, f\rangle) & = & \langle i, q\rangle \\
match(\langle i, q, \langle r, q'\rangle\rangle, \langle i, q, f\rangle) & = & \begin{cases} r' & r' \in \mathbb{E} \\ \langle i, q, \langle r, r'\rangle\rangle & r' \in \mathbb{R} \end{cases} \\
 & & \text{where } r' = match(q', f(r))
\end{array}
$$

FIGURE 4: Application in the Tree Model

Definition 4.2. (*Order-extensionality*) Let L be a language conforming to the syntax of the typed lambda calculus with constants. A model $\mathcal{M}$ interpreting L is *order-extensional* iff for all $f, g \in \mathbb{D}^{\tau\to\sigma}$, $\mathsf{apply}(f, d) \sqsubseteq \mathsf{apply}(g, d)$ for all $d \in \mathbb{D}^\tau$ implies $f \sqsubseteq g$. ∎

Theorem 4.3 *The Tree model for SPCF is order-extensional.*

Proof. Assume $\mathsf{apply}(f, d) \sqsubseteq \mathsf{apply}(g, d)$ for all $d \in \mathbb{D}^\tau$, but $f \not\sqsubseteq g$. Recall that $f = \bigsqcup\{f' \mid f' \sqsubseteq f,\ f' \text{ finite }\}$, $g = \bigsqcup\{g' \mid g' \sqsubseteq g,\ g' \text{ finite }\}$. Thus, there exists a finite $f' \sqsubseteq f$ such that $f' \not\sqsubseteq g$. Hence, for all finite $g' \sqsubseteq g$, it is also true that $f' \not\sqsubseteq g'$. By the following lemma, there exists a finite d such that $\mathsf{apply}(f', d) \not\sqsubseteq \mathsf{apply}(g', d)$ for all $g' \sqsubseteq g$. Since $\mathsf{apply}(g, d) = \bigsqcup\{\mathsf{apply}(g', d) \mid g' \sqsubseteq g\}$, we also have $\mathsf{apply}(f', d) \not\sqsubseteq \mathsf{apply}(g, d)$ by the finiteness of $\mathsf{apply}(f', d)$. Since apply is monotonic, $\mathsf{apply}(f, d) \not\sqsubseteq \mathsf{apply}(g, d)$, which contradicts our assumption. ∎

Thus, the extensionality of arbitrary decision trees reduces to the extensionality of finite decision trees.

Lemma 4.4 *Let f, g be finite elements in $\mathbb{D}^{\tau\to\sigma}$. If for all finite $d \in \mathbb{D}^\tau$, $\mathsf{apply}(f, d) \sqsubseteq \mathsf{apply}(g, d)$ then $f \sqsubseteq g$.*

Proof Idea. The proof proceeds by induction on the depth of the tree g. The induction step is a case analysis on the possible differences between corresponding nodes of f and g. The error elements in the domain play a crucial role in the proof. If f and g are identical except for one node where they make different queries about d (the first argument), we can construct an input that places error_1 in the position specified by f and error_2 in the position specified by g. ∎

The extensionality theorem implies that the domain $\mathbb{D}^{\tau\to\sigma}$ is isomorphic to a domain of *functions* from $\mathbb{D}^\tau$ to $\mathbb{D}^\sigma$ defined as follows.

Definition 4.5. ($\mathbb{F}^{\tau\to\sigma}$) For each f in $\mathbb{D}^{\tau\to\sigma}$, let $fun(f)$ denote the function $\underline{\lambda} x : \mathbb{D}^\tau.\mathsf{apply}(f, x)$. The domain $\mathbb{F}^{\tau\to\sigma}$ is the set $\{fun(f) \mid f \in \mathbb{D}^{\tau\to\sigma}\}$ under the pointwise ordering on functions: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x in $\mathbb{D}^\tau$. ∎

Corollary 4.6 $\mathbb{D}^{\tau\to\sigma}$ *is isomorphic to* $\mathbb{F}^{\tau\to\sigma}$.

ordering, in which successive nodes that are behaviorally indistinguishable collapse. Under this ordering, the model based on a single error is still order-extensional but at the cost of making the ordering ineffective.

4.3 Assigning Meaning to Terms

To assign meaning to phrases in SPCF, we translate programs into a sublanguage without λ-abstractions but with combinators S, K, I. The translation is based on the conventional abstraction algorithm (see Figure 3):

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

After the conversion, it suffices to interpret the primitive functions and combinators. Figure 5 contains pictorial and formal definitions for the primitive combinators.

The definition of the other combinators is tedious. We explain the general idea by defining the denotation of K. Recall that, in a typed setting, K is supposed to represent the abstraction

$$\lambda x_1^{\tau_1\to\ldots\tau_n\to\sigma}.\lambda x_2^\sigma.(\lambda x_3^{\tau_1}\ldots x_{n+2}^{\tau_n}.x_1 x_3 \ldots x_{n+2})$$

Thus, K begins the evaluation process by probing the first argument, 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 about its ith argument, K directs this query to x_{i+2}, ignoring x_2. The answer of x_{i+2} to this query is the information that x_1 needs to know. Thus, after obtaining a response from x_{i+2}, K probes x_1 again with a new query that extends of x_1's response with the additional information provided by x_{i+2}. The next response by x_1 is subjected to the same case analysis as the first one.

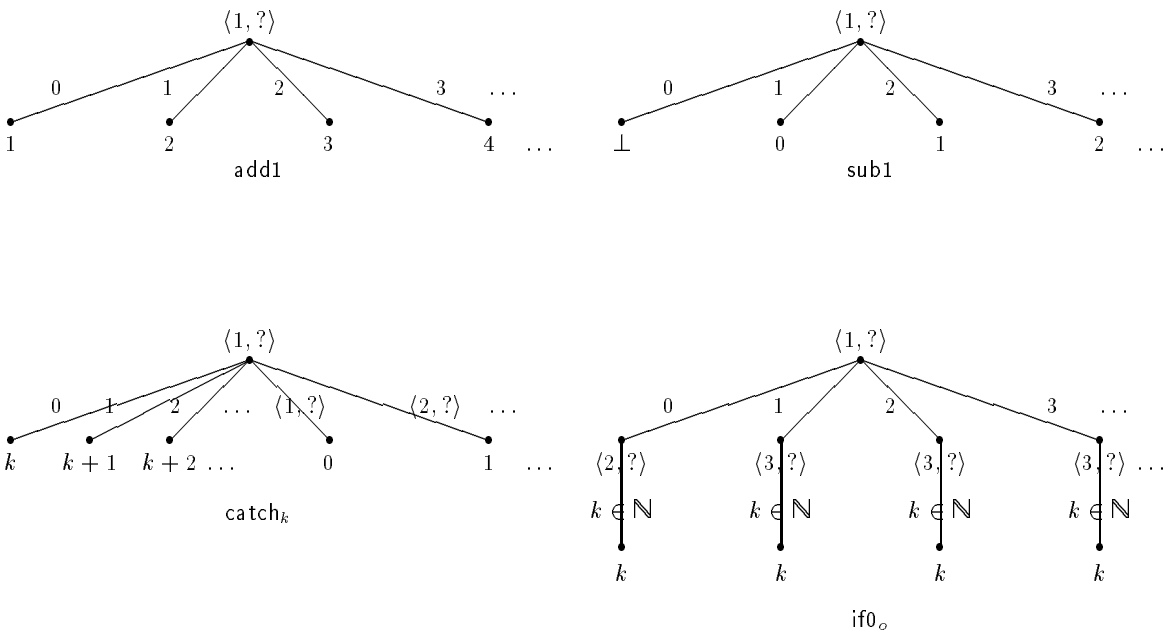
We can formalize this process by defining a chain of finite approximations that encode the recursive process:

$$
\begin{array}{rcl}
\mathsf{K}_0(q)(r) & = & \perp \\
\mathsf{K}_{n+1}(q)(q[?/a]) & = & a \qquad \text{if } a \in \mathbb{N} \\
\mathsf{K}_{n+1}(q)(q[?/\langle i, p\rangle]) & = & \langle i+2, p, \lambda r'.\langle 1, q[?/n], f\rangle\rangle \\
 & & \text{where} \\
 & & \quad f = \mathsf{K}_n(q[?/n]) \\
 & & \quad n = \langle i, p, \langle r', ?\rangle\rangle
\end{array}
$$

K is the limit of these approximations:

$$\mathsf{K} = \bigsqcup\{\langle 1, ?, \mathsf{K}_n(?)\rangle \mid n \in \mathbb{N}\}$$

It is easy to check that $fun(\mathsf{apply}(\mathsf{K}, f))$ is like f except that the new element contains a node of the shape $\langle i+1, q\rangle$ where f contained a node of the shape $\langle i, q\rangle$.



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

FIGURE 5: Primitive Combinators

I and S are defined along the same lines as K. Y can be defined in the usual manner:

$$\mathsf{Y}M = \bigsqcup \{\mathsf{apply}^n(M, \bot) \mid n \in \mathbb{N}\}.$$

4.4 Properties of the Model

The combinators K and S satisfy the usual equations.

Lemma 4.7

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

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

Proof. By induction on n for each approximation K_n and S_n. ∎

This implies that the β axiom for bracket abstractions is valid in this model [1].

Corollary 4.8 ((β)) *For combinator terms M, M', $(\lambda^* x.M)M' = M[x \leftarrow M']$.*

To determine the important equations for catch and error values, we draw on our operational intuition about capturing the behavior of control operators with reduction rules [10, 11]. The main idea is to formalize the notion of a program counter for program text, that is, a tool that determines which term in a program must be evaluated next in order to determine the program's answer. We do this by introducing the auxiliary notion of *evaluation context*:

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

An evaluation context is a context whose hole is in "leftmost-outermost" position, that is, the path from the root to the hole is such that only strict primitive functions are to the left, including applications of catch to an abstraction. If an evaluation context contains an error or bottom value in the hole, it denotes error or bottom.

Lemma 4.9 *For all evaluation contexts E, numbers k, and variables x_1 through x_n,*

$$
\begin{aligned}
E[\mathsf{error}_j] &= \mathsf{error}_j \\
\mathsf{catch}(\lambda^* x_1, \ldots, x_n.E[x_j]) &= j - 1 \\
\mathsf{catch}(\lambda^* x_1, \ldots, x_n.k) &= k + n
\end{aligned}
$$

Proof. We can show that

$$(\lambda^* x_1, \ldots, x_n.E[x_j]) = \langle j, ?, f\rangle$$

for an appropriate branching function f by induction on the structure of the context. The rest follows directly. ∎

The equations for catch, the natural equations for the numerical primitives, and β completely determine an operational reduction semantics for SPCF [10, 11]. The equations also suffice to show in the representability lemma of the following section that certain terms represent given finite elements.

5 The Full Abstraction Theorem

Plotkin [16] showed that an order-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 carries over to SPCF. Since we 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 fully abstract for SPCF.

Theorem 5.1 (Full Abstraction of $\mathcal{T}$) *If e and e' are closed SPCF expressions, then $e \equiv_{\mathcal{T}}$ iff $e \simeq e'$.*

Proof. The proof follows exactly the same outline as Plotkin's proof of the full abstraction theorem for the continuous function model of PCF with parallel operations [16:240]. It relies on the following lemma. ∎

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

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

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

Since the induction step decomposes the element d into a root, a set of proper responses, and an attached set of *sub*trees, we need to prove a stronger result than the lemma because subtrees of d are not necessarily trees in $\mathbb{D}^\tau$. Specifically, we must prove that every finite *sub*tree e of a finite element d in $\mathbb{D}^\tau$ is "representable" in SPCF.

A subtree e is *representable* iff there is a term M and a query p such that (i) $p[?/e]$ is an element of $\mathbb{D}^\tau$ and (ii) $\mathcal{E}[\![M]\!] = p[?/e]$. This definition of subtree representability reduces to tree representability when e is a decision tree and p is empty.

To prove that some term represents a subtree, it is convenient to rely on an equational characterization of the representability criterion. By extensionality, condition (ii) above is equivalent to the constraint that

$$\mathsf{apply}(\mathcal{E}[\![M]\!], d_1, \ldots, d_k) = \mathsf{apply}(p[?/e], d_1, \ldots, d_k)$$

for all $d_1, \ldots, d_k$ in the appropriate domains. Since p determines a context ρ_p for the subtree e, we restate this condition more succinctly. If the arguments $d_1, \ldots, d_k$ satisfy the constraints, $d_i \sqsupseteq \bigsqcup \rho_p(i)$ for all i, then the righthand-side of the equation above is equivalent to $\mathsf{apply}(e, d_1, \ldots, d_k)$. If an argument d_i violates the constraints $\{d_i \not\sqsupseteq \bigsqcup \rho_p(i)\}$, then the subtree e does not affect the result of the application. Consequently, the SPCF expression M represents subtree $e \in \mathbb{D}^\tau(\rho)$ iff

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

for all arguments $d_1, \ldots, d_k$ of appropriate type such that $d_i \sqsupseteq \bigsqcup \rho(i)$.

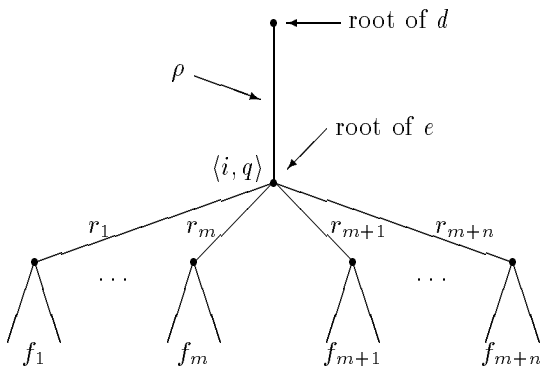
We are now ready to prove that all subtrees of type τ are representable. Since the term M must be a procedure of type τ, it has the form

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

where B is an SPCF expression with free variables $x_1, \ldots, x_k$. The construction of B (and the proof that M represents e) proceeds by induction on the depth of subtrees. The remainder of the proof is a nested induction argument so our induction hypothesis holds for all subtrees of type less than τ and for all smaller subtrees of type τ.

In the base case, e is a leaf in $\mathbb{N}^e_\perp$. It is easy to verify that for any leaf e other than $\perp$, the expression $M = \lambda x_1, \ldots, x_k.e$ represents e. For $e = \perp$, the expression $M = \lambda x_1, \ldots, x_k.\mathsf{Y}(\lambda x.x)$ works similarly.

In the inductive case, e is a finite subtree $\langle i, q, f\rangle$ and ρ is an arbitrary tree context such that $e \in \mathbb{D}^\tau(\rho)$ below



Let the proper domain of the branching function f be the set $\{r_1, \ldots, r_{m+n}\}$ where the responses $r_1, \ldots, r_m$ end in pairs $\langle h_1, p_1\rangle, \ldots, \langle h_m, p_m\rangle$ and the responses $r_{m+1}, \ldots, r_{m+n}$ end in the final answers $a_1, \ldots, a_n$. We refer to the former as final responses and to the latter as query responses. Each edge labeled with r_i points to a subtree f_i whose root node is not bottom. Let $f_1, \ldots, f_n$ denote the subtrees of e corresponding to the responses $r_1, \ldots, r_n$. By the induction hypothesis, each subtree f_i is representable by an SPCF expressions F_i.

The execution of procedure $M = \lambda x_1 \ldots x_k.B$ must first determine the contents of the node N_q identified by q in the value bound to x_i. More precisely, M must determine which, if any, of the responses r_i, that the node N_q matches. Given this information, B can apply F_i (the expression representing the subtree f_i) to the arguments $x_1, \ldots, x_k$.

To understand the process of extracting the node N_q from x_i, we need to state some assumptions. Let ρ_q be the tree context determined by the responses embedded in q. Let the type τ_i of x_i be $\sigma_1 \to \ldots \sigma_l \to o$. Naïvely, we might construct an expression B that tries to obtain the contents of node N_q by applying x_i to a tuple of argument expressions $A_1, \ldots, A_l$ that encode the information in q about x_i's arguments: $\bigsqcup \rho_q(i)$. If N_q is a terminal node, then this naïve strategy works. But it fails when N_q is a query node, because x_i does not return to B until it has gathered enough information from its arguments to produce a final answer. Since $A_1, \ldots, A_l$ only contain the information in ρ_q, x_i will not be able in general to gather enough information to return a final answer. If x_i dominates one of the query responses, say r_j, it will eventually attempt to probe some argument A_h with p_j. When such an event occurs, the naïve argument A does not contain the requested information and the query process will diverge. To solve this problem, we need to embed code in A that escapes to B when this information is requested.

In SPCF, a program can perform non-local exits by applying the catch operator to an appropriate procedure. For our problem, we must define a λ-abstraction M' of m arguments that requests the jth argument precisely when x_i dominates the query responses r_j. Let $y_1, \ldots, y_m$ be the parameters of M'. The key steps in writing the code M' are

1. to associate each query response r_j with the parameter y_j,
2. to embed variable y_j in the argument A_h if r_j ends in $\langle h, p_j \rangle$, and
3. to apply x_i to these arguments.

Hence, M' has the form

$$\lambda^* y_1, \ldots, y_m . x_i \, A_1 \; \ldots \, A_l$$

where the parameters $y_1, \ldots, y_m$ appear free in the argument expressions $A_1, \ldots, A_l$. The precise form of the expressions $A_1, \ldots, A_l$ depends on the query q. For the moment, let us assume that we can construct the argument expressions $A_1, \ldots, A_l$.

The application of catch to the λ-abstraction M' identifies which proper response r_j (if any) that x_i produces for query q. Given the value j, M can invoke the code F_j representing the subtree f_j by performing a case split. Without loss of generality, we assume that the final answers are sorted: $a_1 < a_2 < \ldots < a_n$. Then M is:

$$\begin{array}{l}
\lambda^* x_1^{\tau_1} \ldots x_k^{\tau_k}. \\
\quad \textbf{let } w = \mathsf{catch}(\lambda^* y_1, \ldots, y_m . x_i A_1 \ldots A_l) \textbf{ in} \\
\qquad (\mathsf{if0}\ w\ F_1 x_1 \ldots x_k \\
\qquad \ldots \\
\qquad (\mathsf{if0}\ (\mathsf{sub1}^{m-1}\ w)\ F_m x_1 \ldots x_k \\
\qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_1+m}\ w)\ F_{m+1} x_1 \ldots x_k \\
\qquad \ldots \\
\qquad (\mathsf{if0}\ (\mathsf{sub1}^{a_n+m+m}\ w)\ F_{m+n} x_1 \ldots x_k \\
\qquad \Omega) \ldots)) \ldots)
\end{array}$$

The notation $(\mathsf{sub1}^b\ w)$ denotes the b-fold application of sub1 to w.

By the preceding argument, we have reduced the task of constructing M to constructing the argument expressions $A_1, \ldots, A_l$. Let i be an index in the range $1, \ldots, l$. The construction of A_i falls into one of two cases. In the first case, no query response r_j asks for more information about the ith argument. In this case, we let A_i be the code representing the tree $\bigsqcup \rho_q(i)$, which exists by the induction hypothesis because the type σ_i of A_i is smaller (shallower) than τ.

In the more interesting case, there is some number s of query responses for index i below the root of e. Assume without loss of generality that these query-responses are $r_1 = q[?/\langle h_1, p_1 \rangle], \ldots, r_s = q[?/\langle h_s, p_s \rangle]$. We know that the queries $p_1, \ldots, p_h$ extend the information in $\bigsqcup \rho_q(i)$. Consequently, we can form the tree

$$d_i = \bigsqcup \rho_q(i) \sqcup p_1[?/\langle t+1, ? \rangle] \ldots \sqcup p_h[?/\langle t+b, ? \rangle]$$

where t denotes the arity of type σ_i (the type of A_i). By the induction hypothesis, we can represent this finite tree by an expression A_i' because the type of d_i is smaller than τ. Without loss of generality, we can assume that $y_1, \ldots, y_s$ are the variables that are associated with the queries $p_1, \ldots p_s$. Thus, we can define the expression A_i as follows:

$$A_i = \lambda z_1, \ldots, z_t . A_i' z_1 \ldots z_t \; y_1 \; \ldots \; y_s .$$

This completes the proof sketch. This sketch omits a proof that the term M indeed represents the subtree e, but given the term, this proof can be constructed based on the equations about catch in the previous section. For a more detailed proof, we refer the reader to our technical report [7]. ∎

6 Observable Sequentiality in PCF

It is easy to show that SPCF is sequential. Let $C[M_1, \ldots, M_k]$ be a program satisfying the hypothesis of Definition 2.2, that is, $\mathcal{E}[\![C[M_1, \ldots, M_k]]\!] = n$ and $\mathcal{E}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{E}[\![\Omega]\!]$. We must show that there is one argument position j that forces divergence. Since the model satisfies the equation β, we have:

$$\begin{array}{l}
C[M_1, \ldots, M_k] = \\
\quad \mathsf{apply}(\lambda^* x_1 \ldots x_k . C[x_1, \ldots, x_k], M_1, \ldots, M_k).
\end{array}$$

Hence, it suffices to consider the denotations of 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 $\mathbb{N}^e_\perp$ 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 2.2. The second case specifies that the k-ary procedure probes its jth argument first. But this fact implies that $C[\ ]_1 \ldots [\ ]_k$ diverges if the expression in jth hole diverges—regardless of the expressions in the other holes. Hence, the program behaves sequentially.

But 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 6.1. (*Error-Sensitivity*) A language L based on the typed λ-calculus 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 such that $\mathcal{E}[\![C[M_1, \ldots, M_k]]\!] = n$ for some $n \in \mathbb{N}$ yet $\mathcal{E}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{E}[\![\Omega]\!]$. Then there exists j such that $\mathcal{E}[\![C[M'_1, \ldots, M'_{j-1}, E_i, M'_{j+1}, \ldots, M'_k]]\!] = \mathcal{E}[\![E_i]\!]$ for all $M'_l, 1 \leq l \leq k$. ∎

Theorem 6.2 *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 6.3 *If a language L is error-sensitive, then it is sequential.*

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

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

and

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

for all M'_i. By monotonicity,

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

and

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

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

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

which is precisely what sequentiality demands. ∎

The error-sensitivity of SPCF implies that the tree model is extensional. The domain of decision trees of type $\sigma \longrightarrow \tau$ 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 6.4. (*Observable Sequentiality*) An error-sensitive language L is *observably sequential* iff it satisfies the following property. Let M_1 through M_k be closed expressions and let $C[M_1, \ldots, M_k]$ be a program such that $\mathcal{E}[\![C[M_1, \ldots, M_k]]\!] = n$ for some $n \in \mathbb{N}$ yet $\mathcal{E}[\![C[\Omega, \ldots, \Omega]]\!] = \mathcal{E}[\![\Omega]\!]$. Then there exists a program context $D[\ ]$ such that $\mathcal{E}[\![D[\lambda x_1 \ldots x_k.C[x_1, \ldots, x_k]]]\!] = \mathcal{E}[\![j]\!]$. ∎

The catch procedures make SPCF observably sequential.

Theorem 6.5 *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. In their work on sequential algorithms, Berry and Currien defined an observably sequential language called CDS0 [4]. The sequential algorithms model for CDS0 is fully abstract, but it is not extensional or error-sensitive. Obviously, PCF and PPCF (PCF with parallel operations) are not observably sequential. We conjecture that our construction of a fully abstract model carries over to other observably sequential programming languages.

Since practical programming languages typically contain non-local control operators that permit program evaluation order to be observed, our construction of a fully abstract model for SPCF solves the full abstraction problem for a large class of sequential functional languages. However, it does not yield fully abstract models for sequential languages that include control delimiters [9] (such as **prompt**). The inclusion of control delimiters in a language prevents it from being *error-sensitive*, because delimiters swallow all errors generated within their dynamic extent. Consequently, there

is still an important class of sequential functional languages for which the full abstraction problem remains open.

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. Steve Brookes pointed out a mistake in our original definition of sequentiality.

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. *Modèles complètement adéquats et stables des lambda-calculus typé.* Ph.D. dissertation, Université Paris VII, 1979.
3. Berry, G. and P-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.* **20**, 1982, 265–321.
4. 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.
5. 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.
6. Cartwright, R. and A. Demers. The topology of program termination. In *Proc. Symposium on Logic in Computer Science*, 1988, 296–308.
7. Cartwright, R.S. and M. Felleisen. Observable sequentiality and full abstraction. Technical Report TR91-167. Rice University Department of Computer Science, 1991.
8. Curien, P-L. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Research Notes in Theoretical Computer Science. Piman, London. 1986.
9. Felleisen, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 180–190.
10. 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.*, 1991, to appear.
11. Felleisen, M., D. 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.
12. Kahn, G. and G. Plotkin. Structures des donnés concrètes. INRIA Report 336. 1978.
13. 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.
14. Milner, R. Fully abstract models of typed λ-calculi. *Theor. Comput. Sci.* **4**, 1977, 1–22.
15. Mulmuley, K. *Full Abstraction and Semantic Equivalences.* Ph.D. dissertation, Carnegie Mellon University, 1985. MIT Press, Cambridge, Massachusetts, 1986.
16. Plotkin, G.D. LCF considered as a programming language. *Theor. Comput. Sci.* **5**, 1977, 223–255.
17. 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.
18. Scott, D. S. Domains for denotational semantics. In *Proc. International Conference on Automata, Languages, and Programming*, Lecture Notes in Mathematics 140, Springer-Verlag, Berlin, 1982.
19. Scott, D.S. *Lectures on a Mathematical Theory of Computation.* Techn. Monograph PRG-19, Oxford University Computing Laboratory, Programming Research Group, 1981.
20. 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.
21. Steele, G.L., Jr. *Common Lisp—The Language.* Digital Press, 1984.
22. Steele, G.L., Jr. and G.J. Sussman. The revised report on Scheme, a dialect of Lisp. Memo 452, MIT AI-Lab, 1978.
23. Stoughton, A. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Piman, London. 1986.