A Relational Account of Call-by-Value Sequentiality

Jon G. Riecke
Anders B. Sandholm

BRICS Report Series RS-97-41

ISSN 0909-0878 December 1997

A Relational Account of Call-by-Value Sequentiality

Jon G. Riecke
Bell Laboratories
Lucent Technologies
riecke@bell-labs.com

Anders Sandholm
BRICS[*], *Department of Computer Science*
University of Aarhus, *Denmark*
sandholm@brics.dk

Abstract

We construct a model for FPC, a purely functional, sequential, call-by-value language. The model is built from partial continuous functions, in the style of Plotkin, further constrained to be uniform with respect to a class of logical relations. We prove that the model is fully abstract.

1 Introduction

The problem of finding an abstract description of sequential functional computation has been one of the most enduring problems of semantics. The problem dates from a seminal paper of Plotkin [25], who pointed out that certain elements in Scott models are not definable. In Plotkin's example, the

[*]Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

function

$$por(x, y) = \begin{cases} \mathit{true} & \text{if } x \text{ or } y = \mathit{true} \\ \mathit{false} & \text{if } x \text{ and } y = \mathit{false} \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ denotes divergence, cannot be programmed in the language in the language PCF, a purely functional, sequential, call-by-name language with booleans and numbers as base types. The problem is called the "sequentiality problem" because, intuitively, the only way to program *por* involves evaluating boolean expressions in parallel.

There are, of course, plenty of elements in the Scott model of PCF that cannot be programmed: the domains have uncountably many elements. Nevertheless, the *por* function is worse: it causes two terms in the language PCF to be distinct denotationally even though the terms cannot be distinguished by any program. When similar examples do not exist—in other words, when denotational approximation coincides with operational approximation—the denotational model is said to be **fully abstract**. Shortly after Plotkin's paper, Milner proved that there was exactly *one* fully abstract model of PCF meeting certain conditions [16]. Until recently, all descriptions of this fully abstract model have used operational semantics (see, for instance, [19, 34]). New constructions using games semantics [2, 10, 20] and logical relations [22, 33] have yielded a more abstract understanding of PCF.

The sequentiality problem is enduring because it is robust. For instance, changing the reduction strategy of PCF from *call-by-name* to *call-by-value* makes no difference: versions of the "parallel or" function reappear in the standard Scott model (albeit at higher type). Even for languages that lack an explicit base type, *e.g.*, the polymorphic λ-calculus with recursion, the known Scott models contain parallel elements that cause a failure of full abstraction.

This paper extends the logical-relations approach to another setting. It constructs a model for a *call-by-value*, purely functional, sequential language called FPC. FPC includes a base type with one convergent value, strict products, strong (categorical) sums, functions, and recursive types. Full abstraction for FPC is interesting for at least two reasons:

1. FPC can be regarded as the purely functional, non-polymorphic sublanguage of Standard ML [17]: recursive types and sums are the basis

of `datatype` declarations, and both Standard ML and FPC are call-by-value. By studying FPC, we learn more about programming languages like Standard ML.

2. FPC can serve as an expressive metalanguage for denotational semantics [8, 26]. FPC, for instance, has enough expressive power to encode a call-by-value version of PCF (the base type of numbers can be encoded via a recursive type). Given a fully abstract translation [29] from a language into FPC, the model of FPC yields a fully abstract model of the language.

The relations used in the construction of the model for FPC tease apart the structure of Sieber's relations for PCF [33]. Sieber's model of PCF, and the fully abstract model of PCF using Kripke relations [22], begin from a class of relations at any flat base type (*e.g.*, the partial order of natural numbers with a divergent $\bot$ element below every other element). Sieber's definition of the relations is simple to state.

Definition 1 Suppose $A \subseteq B \subseteq \{1, \dots, n\}$, and let $S^n_{A,B}$ be

$$\{(d_1, \dots, d_n) \mid (\forall i \in A.\ d_i \neq \bot) \implies (\forall i, j \in B.\ d_i = d_j)\}.$$

Then R is an n-ary **sequentiality relation** if R is the intersection of relations of the form $S^n_{A,B}$.

These relations thus have an elegant *semantic* definition: nothing in the definition refers to the terms or operations of PCF. They also seem to say something about sequential computation: if certain elements of a tuple must converge, then other elements must converge. (It is helpful to think of the elements of a relation as potential results of a function.) One can easily show all PCF terms preserve the sequentiality relations, and that *por* does not preserve the sequentiality relation $S^3_{\{1,2\},\{1,2,3\}}$. It must therefore be the case that *por* is not definable.

Sieber's definition of "sequentiality relation" seems to be limited to flat base types. It does not make sense to check for equality at functional type, and the relations say nothing about more complicated partial order structures. It is difficult to see how, for instance, to extend directly the definition to complex sums such as $\mathbf{int} \oplus (\mathbf{int} \Rightarrow \mathbf{int})$.

Instead of directly extending Sieber's relations, our relations break down sequentiality relations into two components. The first captures the sequential behavior of *termination*: if certain elements in a tuple in the relation terminate (are non-bottom), then certain other elements in the tuple must terminate. The second captures the behavior of *sums*: if certain elements in a tuple lie in one side of a sum type, other components must lie in the same side of the sum type. This is much like Sieber's definition in asking for *equality* of all B-indexed elements of a tuple in $S^n_{A,B}$. The interesting case comes when this second component of relations is lifted to types other than sums: when, for example, the tuple is a tuple of elements in a function type. In essence, the second component of relations encodes a form of "computation tree," stating which subtuples of a tuple form consistent traces of the computation so far.

We begin by introducing the language FPC, the main language of study in this paper. We then describe the form of the relations, showing the decomposition into the two portions described above. We follow the O'Hearn-Riecke construction for PCF [22] and lift the relations to *Kripke relations*, using a definition found in [13]. It will be these relations that will be used to define a category in which a model of FPC can be constructed. The Kripke relations will be used to establish the full abstraction of the model.

2 The Language FPC

FPC as described in [8, 26] is a call-by-value, purely functional language with a single base type **unit**, sums, products, functions, and recursive types. More familiar base types, such as booleans or natural numbers, can be easily constructed in the language.

FPC has types given by the grammar

$$s, t ::= \mathbf{unit} \mid (s \oplus t) \mid (s \otimes t) \mid (s \Rightarrow t) \mid \alpha \mid (\mathbf{rec}\ \alpha.\ t)$$

where α ranges over a collection of **type variables**. Types are identified up to renaming of type variables bound by **rec**. The raw terms of FPC are given by the grammar

$$\begin{array}{rcl} M, N, P & ::= & x \mid (\lambda x : t.\, M) \mid (M\ N) \mid \\ & & \langle\rangle \mid \langle M, N\rangle \mid (\mathbf{proj}_i\ M) \mid (\mathbf{inj}_i\, M) \mid \\ & & (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{inj}_1(x).N\ \mathbf{or}\ \mathbf{inj}_2(x).P) \mid \\ & & (\mathbf{intro}_{\mathbf{rec}\ \alpha.\ s}\ M) \mid (\mathbf{elim}_{\mathbf{rec}\ \alpha.\ s}\ M) \end{array}$$

A typing judgement is a formula of the form $\Gamma \vdash M : t$ where M is a term, t a type, and Γ is a **typing context**, *i.e.*, a finite function from variables to types. Rules for deriving typing judgements appear in Table 1. In the type rules, we assume that all types are closed.

Evaluation rules for FPC appear in Table 2. In the rules, we use the notation $M[N/x]$ to denote capture-free substitution of N for x in M. Notice that function application in FPC is call-by-value: arguments to functions must be values before they are substituted into bodies of functions. The operational approximation relation can then be defined as follows:

Definition 2 *$M \sqsubseteq_{FPC} N$ if for any context $C[\cdot]$ such that $C[M]$ and $C[N]$ are closed, well-typed terms, $C[M] \Downarrow V$ implies $C[N] \Downarrow V'$ for some V'.*

Note that we observe termination at any type.

FPC is a very sparse language: there is no recursion nor even any obvious divergent computations. Nevertheless, it still has enough computing power for many applications. For instance, Plotkin [26] and Gunter [8] show how to build recursion operators using recursive types. For a slightly simpler example, one can encode a sequencing operation: $(M; N)$ stands for the term $((\lambda x : s.\, N)\ M)$, where x does not occur free in N. Indeed, the semantics of many programming languages—including non-functional languages—can be given by translation to FPC. FPC's main deficiency as a *metalanguage* for denotational semantics is a lack of parametric polymorphism (as in the Girard–Reynolds calculus [7, 27]), which precludes a good representation of abstract data types.

3 Category of Meanings

In this section we construct a category suitable for interpreting FPC. Throughout the section, **DCPO** denotes the category of dcpos and partial continuous functions, where a dcpo is a directed-complete poset (not necessarily possessing a least element).

The category is built from objects that have both dcpo structure and relational structure. The relations are defined in two stages. First, we show how to define base relations suitable for modeling the language. Second, we lift the relations to *Kripke relations* of varying arity. The category of meanings will use Kripke relations; morphisms will be continuous functions that additionally preserve the Kripke relational structure.

Table 1: Type Rules for FPC.

$$\Gamma, x : t \vdash x : t$$

$$\Gamma \vdash \langle\rangle : \mathbf{unit}$$

$$\frac{\Gamma, x : s \vdash M : t}{\Gamma \vdash (\lambda x : s.\ M) : (s \Rightarrow t)}$$

$$\frac{\Gamma \vdash M : (s \Rightarrow t) \qquad \Gamma \vdash N : s}{\Gamma \vdash (M\ N) : t}$$

$$\frac{\Gamma \vdash M : s \qquad \Gamma \vdash N : t}{\Gamma \vdash \langle M, N\rangle : (s \otimes t)}$$

$$\frac{\Gamma \vdash M : (s_1 \otimes s_2)}{\Gamma \vdash (\mathbf{proj}_i\ M) : s_i}$$

$$\frac{\Gamma \vdash M : s_i}{\Gamma \vdash (\mathbf{inj}_i\, M) : (s_1 \oplus s_2)}$$

$$\frac{\Gamma \vdash M : (s_1 \oplus s_2) \qquad \Gamma, x : s_i \vdash N_i : t}{\Gamma \vdash (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{inj}_1(x).N_1\ \mathbf{or}\ \mathbf{inj}_2(x).N_2) : t}$$

$$\frac{\Gamma \vdash M : s[\mathbf{rec}\ \alpha.\ s/\alpha]}{\Gamma \vdash (\mathbf{intro}_{\mathbf{rec}\ \alpha.\ s}\, M) : (\mathbf{rec}\ \alpha.\ s)}$$

$$\frac{\Gamma \vdash M : (\mathbf{rec}\ \alpha.\ s)}{\Gamma \vdash (\mathbf{elim}_{\mathbf{rec}\ \alpha.\ s}\, M) : s[\mathbf{rec}\ \alpha.\ s/\alpha]}$$

Table 2: Evaluation Rules for FPC.

$$\langle\rangle \Downarrow \langle\rangle$$

$$(\lambda x : s.\, M) \Downarrow (\lambda x : s.\, M)$$

$$\frac{M \Downarrow (\lambda x : s.\, M') \qquad N \Downarrow V' \qquad M'[V'/x] \Downarrow V}{(M\ N) \Downarrow V}$$

$$\frac{M_1 \Downarrow V_1 \qquad M_2 \Downarrow V_2}{\langle M_1, M_2\rangle \Downarrow \langle V_1, V_2\rangle}$$

$$\frac{M \Downarrow \langle V_1, V_2\rangle}{(\mathbf{proj}_i\ M) \Downarrow V_i}$$

$$\frac{M \Downarrow V}{(\mathbf{inj}_i\ M) \Downarrow (\mathbf{inj}_i\ V)}$$

$$\frac{M \Downarrow (\mathbf{inj}_i\ V) \qquad N_i[V/x] \Downarrow R}{(\mathbf{case}\ M\ \mathbf{of}\ \mathbf{inj}_1(x).N_1\ \mathbf{or}\ \mathbf{inj}_2(x).N_2) \Downarrow R}$$

$$\frac{M \Downarrow V}{(\mathbf{intro}_{\mathbf{rec}\ \alpha.\ s}\ M) \Downarrow (\mathbf{intro}_{\mathbf{rec}\ \alpha.\ s}\ V)}$$

$$\frac{M \Downarrow (\mathbf{intro}_{\mathbf{rec}\ \alpha.\ s}\ V)}{(\mathbf{elim}_{\mathbf{rec}\ \alpha.\ s}\ M) \Downarrow V}$$

3.1 Base relations

The basic relations of the model come in two varieties: **termination relations** and **computational relations**. Both kinds of relations range over elements of a dcpo. Following [13, 22], it will be helpful to represent relations by *finite, total functions* from indices to values instead of as tuples of values. Of course, there is no real difference between the two presentations, but the functional version will be easier to extend to Kripke relations later on.

Termination relations are defined using simple implicational theories. Let w be a finite set of indices. Then a **w-termination theory** is a set of implications of the form $(d_1, \ldots, d_n \vdash d)$ where $d_1, \ldots, d_n, d \in w$ and $n \geq 1$. Intuitively, each implication states a property similar to the Sieber's sequentiality relations: if a function halts on the indices in the argument, it must halt on the index in the result. Indeed, the termination part of Sieber's relations $S^n_{A,B}$ can be encoded: if $A \subseteq B \subseteq w$, $A = \{d_1, \ldots, d_k\}$, and $B = \{d_1, \ldots, d_k, d_{k+1}, \ldots, d_n\}$ then $S^w_{A,B}$ corresponds to the implications $(d_1, \ldots, d_k \vdash d_{k+1}), \ldots, (d_1, \ldots, d_k \vdash d_n)$.

For $w' \subseteq w$, we say that $w' \models (d_1, \ldots, d_n \vdash d)$ if $d_1, \ldots, d_n \in w'$ implies $d \in w'$. If T is a set of such implications, we say that $w' \subseteq w$ is a **T-model** if $w' \models \psi$ for all $\psi \in T$. There is an alternative characterization of T-models that can be helpful in proving facts about termination relations (a closely-related characterization can be found in [35], page 22). Suppose w is a finite set and $X \subseteq \mathcal{P}(w)$. Then X is a **closure system** if $w \in X$, $\emptyset \in X$, and X is closed under intersection.

Proposition 3 *Suppose w is a finite set. Then $X \subseteq \mathcal{P}(w)$ is a closure system iff there is a w-termination theory T such that X is the set of T-models.*

Proof: For the easy direction, suppose T is a w-theory. Then obviously w and $\emptyset$ are T-models. To see closure under intersection, suppose w_1, w_2 are T-models. Suppose $(d_1, \ldots, d_n \vdash d) \in T$, and $d_i \in w_1 \cap w_2$ for all i. Then $d \in w_1$ and $d \in w_2$, hence $d \in w_1 \cap w_2$. Thus, $w_1 \cap w_2$ is also a T-model.

For the other direction, suppose X is a closure system. Define the theory T by

$$(d_1, \ldots, d_n \vdash d) \in T \text{ iff for all } w' \in X,\ w' \models (d_1, \ldots, d_n \vdash d).$$

Let $Y = \{w' \mid w' \text{ is a } T\text{-model}\}$; we want to show that $X = Y$.

It is obvious that $X \subseteq Y$: if $w' \in X$, then by construction it satisfies all the formulas in T and hence is a T-model. Conversely, suppose $w' \notin X$. Let

$$w_0 = \bigcap \{w_1 \in X \mid w' \subseteq w_1\}.$$

Since X is closed under finite intersections, and the set above is finite because w is, w_0 must be in X. But since $w' \notin X$ and $w' \subseteq w_0$, it must be the case that there is a $d \in w_0$ such that $d \notin w'$. Also, because $w' \notin X$, it cannot be the empty set. Now consider the formula $(w' \vdash d)$. This is a legal formula because w' has at least one element. Note that $(w' \vdash d) \in T$, and $w' \not\models (w' \vdash d)$. Thus, $w' \notin Y$, completing the proof. ■

The proof is similar to a part of Sieber's proof that the sequentiality relations are precisely those that are closed under the operations of PCF (see [33]).

Termination theories are the building blocks of the first kind of relations, the termination relations. Let T be a w-termination theory, and D be a dcpo. A **T-termination relation on** D is a set of the form

$$R \subseteq \bigcup_{w' \text{ is a } T\text{-model}} [w' \to_t D],$$

where $[w' \to_t D]$ denotes the set of all total (set-theoretic) functions from w' to D, such that the following properties hold:

1. **Non-emptiness:** R is non-empty.

2. **Directed completeness:** R is directed complete, where $f \sqsubseteq g$ iff f, g have the same domain w' and for all $d \in w'$, $f(d) \sqsubseteq g(d)$ in D.

3. **Downward closure:** For any $f \in R$ with domain w' and T-model $w'' \subseteq w'$, $(\underline{\lambda} d \in w''.\, f(d)) \in R$.

Here, $(\underline{\lambda} d \in w''.\, f(d))$ stands for the function with domain w'', whose return value is $f(d)$; thus, an element of a termination relation R is a function from a subset of indices to elements of the dcpo. To get some intuition, it is again helpful to think of indices as possible arguments to a function, and the elements as the return values of the function. An element of R then represents a "related" set of values returned by a function.

One subtlety in this definition is the $\sqsubseteq$ relation: we only compare elements of R that have the *same domain*. The definition of $\otimes$ on relations does not produce a directed complete relation if we regard the elements of R as partial functions and compare elements with different domains. A second subtlety arises from the first and third conditions. These two conditions imply that the empty function is always an element of R, which is necessary to prove that divergent terms always respect the relations.

Of course, functions represented by terms in FPC may have much more complex behavior: they may test some of their arguments and branch based on the results. At the ends of branches are "related" values returned by the function, but there need be no relationship between the values returned by different branches. To model this behavior, we use computational relations. We first need a preliminary definition. Let T be a w-termination theory. Then S is a **T-computational theory** if S is a set of finite sets of T-models, *i.e.*, each element of S has the form $\{w_1, \ldots, w_k\}$, and satisfies the following conditions:

- If w' is a T-model, then $\{w'\} \in S$;
- If $\{w_1, \ldots, w_k\}$ and $\{w'_1, \ldots, w'_l\}$ are in S, and $w_{i,j} = (w_i \cap w'_j)$, then $\{w_{1,1}, \ldots, w_{k,l}\} \in S$; and
- If $\{w_1, \ldots, w_k\}$ and $\{w'_1, \ldots, w'_l\}$ are in S, and for all i and some j, $w'_i \subseteq w_j$, then
$$\{w_1, \ldots, w_{j-1}, w'_1, \ldots, w'_l, w_{j+1}, \ldots, w_k\} \in S.$$

The elements of S are called **path sets** for reasons that will become clear in a moment.

The path sets permit the definition of computational relations, which just lifts a termination relation to a partial function on indices. Suppose S is a T-computational theory and R is a T-termination relation on D. Define the **computational relation R_S** by

$$\begin{aligned} R_S = \{\, f \in [w \to_p D] \mid {} & \text{there exists } \{w_1, \ldots, w_k\} \in S \text{ such that} \\ & f(d) \downarrow \text{ iff } d \in w_i \text{ for some } i, \text{ and for all } i, \\ & (\underline{\lambda} d \in w_i.\, f(d)) \in R \ \} \end{aligned}$$

where $[w \to_p D]$ represents the set of *partial* set-theoretic functions from the set w to the set D. The computational relations are reminiscent of Moggi's analysis of call-by-value via monads [18], and they will play a similar role in the semantics below.

It is useful to think of elements of R_S as the interpretation of a term given some environment. A computation branches based on its inputs and returns some final results at the end. Each set in $\{w_1, \ldots, w_n\}$ represents a path in that computational tree; the answers returned at the end of a path must be consistent, hence the restriction of the function to $(\underline{\lambda} d \in w_i.\, f(d))$ must be in R. We can now see from where the three conditions on computational theories come. The first condition says that a potential path set that does no branching is a valid path set; the second says path sets can be combined; the third says that an element of a path set may be replaced by finer-grain path set, which amounts to adding a set of branches to a non-branching part of the computation.

3.2 Extension to Kripke relations

We could build a category of meanings directly using termination and computational theories; we would probably not get a fully abstract model (see the discussion in Section 6). Instead, we extend the relations to Kripke relations of varying arity [13]. Kripke relations of this kind begin from an **index category** whose objects are finite sets and whose morphisms are total functions (not necessarily all of them).

Definition 4 Suppose $\mathbf{C}$ is an index category. A **C-termination theory** is a family

$$T = \{\, T^w \mid w \in \mathrm{Ob}(\mathbf{C}),\ T^w \text{ is a } w\text{-termination theory} \,\},$$

such that, for any $\varphi : v \xrightarrow{\mathbf{C}} w$, if w' is a T^w-model, then $\{\, d \in v \mid \varphi(d) \in w' \,\}$ is a T^v-model.

A Kripke relation is a set of termination relations that must fit together.

Definition 5 Let $\mathbf{C}$ be an index category, T be a $\mathbf{C}$-termination theory, and D be a dcpo. A **$\mathbf{C}, T$-termination relation on D** is a family of sets

$$R = \{R^w \mid w \in \mathrm{Ob}(\mathbf{C}),\ R^w \text{ is a } T^w\text{-termination relation on } D\}$$

satisfying the

> **Kripke monotonicity condition:**
> For any $f \in R^w$ with domain w' and $\varphi : v \xrightarrow{\mathbf{C}} w$, then $(\underline{\lambda} d \in v'.\, f(\varphi(d))) \in R^v$, where $v' = \{\, d \in v \mid \varphi(d) \in w' \,\}$.

Computational theories also extend straightforwardly to the Kripke case:

Definition 6 Let $\mathbf{C}$ be an index category and T be a $\mathbf{C}$-termination theory. Then S is a **$\mathbf{C}, T$-computational theory** if S is a set, indexed by objects w of $\mathbf{C}$, of T^w-computational theories.

If S is a $\mathbf{C}, T$-computational theory and R is a $\mathbf{C}, T$-termination relation on D, we let R^w_S denote the computational relation built from R^w and S^w.

3.3 A category for interpreting FPC

We now have enough machinery to build the category $\mathcal{SR}$ (for *sequentiality relations*).

- Objects. An object A consists of a dcpo $|A|$ and a $\mathbf{C}, T$-termination relation $A(T, S)$ on $|A|$ for each $\mathbf{C}$-termination theory T and $\mathbf{C}, T$-computational theory S. Objects must also satisfy the

  Concreteness Condition:
  For any $d \in D$, $(\underline{\lambda} i \in w.\, d) \in A(T, S)^w$.

- Morphisms. A morphism $f : A \to B$ is a partial continuous function $f : |A| \rightharpoonup |B|$ satisfying the

  Uniformity Condition:
  For all $\mathbf{C}$, $\mathbf{C}$-termination theories T, $\mathbf{C}, T$-computational theories S, and $h \in A(T, S)^w$, $(h; f) \in B(T, S)^w_S$, where $(h; f)$ denotes diagrammatic composition of h and f.

Composition and identities are inherited from **DCPO**. It is straightforward to check that $\mathcal{SR}$ is indeed a category; the only slightly non-obvious step is checking that composition is uniform, for which one needs the closure properties of path sets. Moreover, the category is dcpo-enriched, meaning that for any objects A, B, the set of morphisms $Hom_{\mathcal{SR}}(A, B)$ is a dcpo; the morphisms are ordered

$$f \sqsubseteq g \text{ iff}$$
$$\text{for any } a \in |A|,\ f(a)\!\downarrow \text{ implies that } g(a)\!\downarrow \text{ and } f(a) \sqsubseteq g(a) \text{ in } |B|$$

where $f(a)\!\downarrow$ means that $f(a)$ is defined.

3.4 Interpretation of FPC

The basic constructions needed for interpreting FPC are the following:

Void object:

$|void| = \{\}$
$void(T,S)^w = \{\emptyset\}$

Unit object:

$|unit| = \{\top\}$
$unit(T,S)^w = \{\, (\underline{\lambda} d \in w'.\, \top) \mid w' \text{ is a } T^w\text{-model}\,\}$

Products:

$|A \otimes B| = |A| \times |B|$ (cartesian product)
$(A \otimes B)(T,S)^w = \{\langle g,h\rangle \mid g \in A(T,S)^w,\ h \in B(T,S)^w,$ and g,h have same domain$\}$

Coproducts:

$|A \oplus B| = |A| \oplus |B|$ (disjoint union)
$(A \oplus B)(T,S)^w = \{\, f \mid (\exists g \in A(T,S)^w. f = (g; inj_1)) \vee (\exists h \in B(T,S)^w. f = (h; inj_2))\,\}$

Exponentiation:

$|A \Rightarrow B| = Hom_{\mathcal{SR}}(A,B)$,
$(A \Rightarrow B)(T,S)^w = \{\, f \mid \forall \varphi : v \xrightarrow{\mathbf{C}} w, g \in A(T,S)^v.\ (\underline{\lambda} d \in v.\, (f(\varphi(d))\,(g(d)))) \in B(T,S)^v_S\}$

The definition of exponentiation on relations follows the one in [13, 22]. Note that the definition of the relational component of *unit* captures much of the intuition we set out for termination relations.

One may check that *void*, *unit* are objects and $\otimes$, $\oplus$, and $\Rightarrow$ are bifunctors (where $\Rightarrow$ is contravariant in the first argument as usual). To interpret FPC, a bit more structure is required. It is not hard to show that $\oplus$ is a coproduct, and the usual constructions on pairs (projections and pairing) also define morphisms. Also, the category is a *partial cartesian closed category* in the sense of [5]. This provides all the structure to interpret FPC types and terms except those involving recursive types.

The interpretation of recursive types requires adapting standard results from domain theory (see, for instance, [1, 8]) to $\mathcal{SR}$. Recursive types are interpreted via a colimit construction. Given objects A, B, f is an **embedding projection pair** (ep-pair for short) if $f = (f^e : A \to B, f^p : B \to A)$, $(f^e; f^p) = id_A$, and $(f^p; f^e) \sqsubseteq id_B$. An **expanding sequence** is a tuple

$$(\{D_n \mid n \geq 0\}, \{f_{mn} \mid f_{mn} : D_n \to D_m \text{ is an ep-pair, } n \leq m\}),$$

such that $f_{nn} = id_{D_n}$ and for any $n \leq k \leq m$, $f^e_{mn} = (f^e_{kn}; f^e_{mk})$ and $f^p_{mn} = (f^p_{mk}; f^p_{kn})$. Given an expanding sequence as above, define the object D by

$$|D| = \{\langle x_0, x_1, \ldots\rangle \mid x_i \in D_i \text{ and } \forall n \leq m,\ x_n = f^p_{mn}(x_m)\}$$

$$\begin{aligned} D(T,S)^w = \{h \in [w' \to_t D] \mid\ & w' \text{ is a } T^w\text{-model, and there exist} \\ & h_i \in D_i(T,S)^w \text{ with domain } w', \\ & \text{such that } h = \langle h_0, h_1, h_2, \ldots\rangle \text{ and} \\ & \text{for all } n \leq m,\ h_n = (h_m; f^p_{mn})\}. \end{aligned}$$

This is indeed an object in $\mathcal{SR}$.
The embedding-projection pairs $(\mu^e_m : D_m \to D, \mu^p_m : D \to D_m)$, defined by

$$\begin{aligned} &\mu^p_m(\langle x_0, x_1, \ldots\rangle) = x_m \\ &\mu^e_m(d) = \left\langle \textstyle\bigsqcup_{k\geq m,0}(f^p_{k0}(f^e_{km}(d))), \bigsqcup_{k \geq m,1}(f^p_{k1}(f^e_{km}(d))), .. \right\rangle \end{aligned}$$

make the object D into a colimit of the expanding diagram. The only non-standard part of the proof lies in showing that μ^e_m and μ^p_m satisfy the uniformity property.

Recursive types make the interpretation somewhat complex [8, 26]. The meaning of a possibly open type is a *functor* from its free variables to the category $\mathcal{SR}$. The functor, though, may be covariant in some of its arguments and contravariant in others. For instance, the type expression $(\alpha \Rightarrow \beta)$ has α occurring contravariantly and β occurring covariantly. We follow recent tradition (see [6, 24]) and convert a type with n-variables into one with $2n$ variables: $\vec{\alpha}$ occurring only negatively and $\vec{\beta}$ occurring only positively. For instance, the type $(\alpha \Rightarrow \alpha)$ gets converted to the type $(\alpha \Rightarrow \beta)$.

The meaning of a type is a functor $(\mathcal{SR}^{op})^n \times (\mathcal{SR})^n \to \mathcal{SR}$. The definition on objects is

$$\begin{aligned} [\![\mathbf{unit}]\!](\vec{A}, \vec{B}) &= \mathit{unit} \\ [\![\mathbf{rec}\ \alpha.\ s]\!](\vec{A}, \vec{B}) &= \mathit{FIX}([\![s]\!](X :: \vec{A}, Y :: \vec{B})) \\ [\![s \oplus t]\!](\vec{A}, \vec{B}) &= ([\![s]\!](\vec{A}, \vec{B}) \oplus [\![t]\!](\vec{A}, \vec{B})) \\ [\![s \otimes t]\!](\vec{A}, \vec{B}) &= ([\![s]\!](\vec{A}, \vec{B}) \otimes [\![t]\!](\vec{A}, \vec{B})) \\ [\![s \Rightarrow t]\!](\vec{A}, \vec{B}) &= ([\![s]\!](\vec{B}, \vec{A}) \Rightarrow [\![t]\!](\vec{A}, \vec{B})) \end{aligned}$$

where FIX is an operation on functors that generates a canonical colimit (it can be easily expressed as a colimit of an expanding sequence starting from the *void* object, using the functor to build the subsequent elements of the chain). We omit the full definition here, and the definition of $[\![s]\!]$ on morphisms. When acting on ep-pairs, the functor $[\![s]\!]$ is continuous in the sense described in [1]. The canonical maps

$$\begin{array}{l} \mathit{intro} : FIX(G(X,Y)) \to G(FIX(G(X,Y)), FIX(G(X,Y))) \\ \mathit{elim} : G(FIX(G(X,Y)), FIX(G(X,Y))) \to FIX(G(X,Y)) \end{array}$$

which define an isomorphism, are used to give meaning to the **intro** and **elim** term constructors.

The meaning of terms can then be given using the combinators of the category. Suppose $\Gamma \vdash M : s$. If

$$\Gamma = x_1 : t_1, ..., x_n : t_n,$$

define $[\![\Gamma]\!] = [\![t_1]\!] \otimes \ldots \otimes [\![t_n]\!]$. (If Γ is empty, $[\![\Gamma]\!]$ is the object *unit*). Then $[\![\Gamma \vdash M : s]\!]$ is a morphism in $\mathcal{SR}$. We omit the full definition here. For notational convenience, if $\emptyset \vdash M : s$, we write $[\![M]\!]$ for the corresponding element $[\![\emptyset \vdash M : s]\!]\top \in [\![s]\!]$.

4 Examples

Even though the semantic category $\mathcal{SR}$ is arguably complicated, it does support simple reasoning about definability of FPC terms.

4.1 Parallel convergence testing is not definable

Consider the partial continuous function

$$f : (\mathit{unit} \Rightarrow \mathit{unit}) \otimes (\mathit{unit} \Rightarrow \mathit{unit}) \to \mathit{unit}$$

defined

$$f\langle g, h\rangle = \begin{cases} \top & \text{if } g(\top)\downarrow \text{ or } h(\top)\downarrow \\ \text{undefined} & \text{otherwise.} \end{cases}$$

f appears to need to do its calculation "in parallel." We would like to prove that f is *not* a morphism in the category, which will immediately imply that f is not definable.

The argument follows the proof of Sieber (also due to Plotkin, see [3]) that *por* is not definable. In this case, we need to exhibit an index category $\mathbf{C}$, a choice of $\mathbf{C}$-termination theory T, and a $\mathbf{C}, T$-computational theory S. Pick $\mathbf{C}$ to be the category with just one index set $w = \{1, 2, 3\}$, T^w to be the theory with just one implication $(1, 2 \vdash 3)$, and S to be the set $\{\,(w') \mid w' \text{ is a } T^w\text{-model}\,\}$. Let $h : \mathit{unit} \rightarrow \mathit{unit}$ be the function that returns $\top$ given $\top$, and $h' : \mathit{unit} \rightarrow \mathit{unit}$ be the empty partial function. Also, let

$$g_1 = \langle h, h' \rangle, \qquad g_2 = \langle h', h \rangle, \qquad g_3 = \langle h', h' \rangle.$$

Returning to standard tuple notation for relations, we claim

$$(g_1, g_2, g_3) \in ((\mathit{unit} \Rightarrow \mathit{unit}) \otimes (\mathit{unit} \Rightarrow \mathit{unit}))(T, S)^w.$$

To prove the claim, one may show $(h, h', h'), (h', h, h') \in (\mathit{unit} \Rightarrow \mathit{unit})(T, S)^w$ by a simple case analysis. But

$$(f(g_1), f(g_2), f(g_3)) \notin \mathit{unit}(T, S)^w_S,$$

so f is not uniform. Thus, f cannot be a morphism, and hence it cannot be definable.

4.2 Sieber's last example is not definable

The second example is essentially the last example in [33], an example modified from [4]. Let

- *bool* be the object $(\mathit{unit} \oplus \mathit{unit})$, and
- $A = (\mathit{unit} \Rightarrow \mathit{bool}) \otimes (\mathit{unit} \Rightarrow \mathit{bool})$.

Let *true* denote $\mathit{inj}_1(\top) \in \mathit{bool}$ and *false* denote $\mathit{inj}_2(\top) \in \mathit{bool}$. Consider the morphisms $g_1, g_2, g_3, g_4 : A \rightarrow \mathit{unit}$ defined by

$$\begin{aligned}
g_1\langle h_1, h_2 \rangle &\simeq \begin{cases} \top & \text{if } h_2(\top) = \mathit{true} \\ \top & \text{if } h_1(\top) = \mathit{true} \text{ and} \\ & \quad h_2(\top) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
g_2\langle h_1, h_2 \rangle &\simeq \begin{cases} \top & \text{if } h_1(\top) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
g_3\langle h_1, h_2 \rangle &\simeq \begin{cases} \top & \text{if } h_2(\top) = \mathit{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
g_4\langle h_1, h_2 \rangle &\simeq \text{undefined}
\end{aligned}$$

We claim that any $f : (A \Rightarrow \mathit{unit}) \to \mathit{unit}$ where

$$f(g_1) = \top, \quad f(g_2) = \top, \quad f(g_3) = \top, \quad f(g_4)\uparrow$$

is not definable in FPC.

The proof of nondefinability requires a nontrivial use of path sets, and follows the ideas in [33]. Pick $\mathbf{C}$ to be the trivial index category $\mathbf{C}$ with object $w = \{1, 2, 3, 4\}$. Let T^w be the termination theory T^w with just one implication $(1, 2, 3 \vdash 4)$. Let S be the set of path sets $\{w_1, \ldots, w_n\}$ such that

1. Each w_i is a T^w-model;
2. If $i \neq j$, then w_i, w_j are disjoint;
3. If $1 \in w_i$ and $2 \in w_j$, then $i = j$; and
4. If $1 \in w_i$ and $3 \in w_j$, then $i = j$.

It is not hard to prove that this is a $\mathbf{C}, T$-computational theory, and that

$$(g_1, g_2, g_3, g_4) \in (A \Rightarrow \mathit{unit})(T, S)^w.$$

However, it is evident that

$$(f(g_1), f(g_2), f(g_3), f(g_4)) \notin \mathit{unit}(T, S)^w_S$$

because of the last two conditions on the path sets. Thus, there is no such definable f.

5 Full Abstraction

We prove full abstraction for FPC by considering a different language called Finite FPC. A model of Finite FPC lives in the same category $\mathcal{SR}$. We prove that all elements of the model of Finite FPC are definable by terms. Terms in Finite FPC have representatives in FPC, which will be used to establish full abstraction.

The proof of full abstraction follows the structure of the proof in [22].

5.1 Finite FPC

Finite FPC is a simplification of FPC. It has types given by the grammar

$$s, t ::= \mathbf{void} \mid \mathbf{unit} \mid (s \oplus t) \mid (s \otimes t) \mid (s \Rightarrow t)$$

Finite FPC thus differs from FPC in not having recursive types and in having **void**. The raw terms are given by the grammar

$$\begin{array}{rcl} M, N, P & ::= & x \mid \Omega \mid (\lambda x : t.\, M) \mid (M\ N) \mid \langle\rangle \mid \\ & & \langle M, N\rangle \mid (\mathbf{proj}_i\ M) \mid (\mathbf{inj}_i\, M) \mid \\ & & (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{inj}_1(x).N\ \mathbf{or}\ \mathbf{inj}_2(x).P) \end{array}$$

Note that there is no recursion on terms in the language, only a divergent term Ω at all types. Rules for deriving typing judgements are as in Table 1 with three exceptions: the rules for **elim** and **intro** are omitted and the rule for divergence

$$\Gamma \vdash \Omega : s$$

is added. The language has the evident interpretation in the category $\mathcal{SR}$ using the objects and morphisms described in the previous section.

5.2 Construction of relation

To prove that all elements of the model of Finite FPC are representable by terms, we will consider a particular index category $\mathbf{C}$, particular $\mathbf{C}$-termination theory T, and particular $\mathbf{C}, T$-computational theory S. Define the index category to be the set of sets of the form

$$[s_1, \ldots, s_n] = |[\![s_1]\!] \otimes \ldots \otimes [\![s_n]\!]|.$$

Morphisms are the projections $[s_1, \ldots, s_{n+k}] \to [s_1, \ldots, s_n]$. For any object $w = [s_1, \ldots, s_n]$ of $\mathbf{C}$, let $s = (s_1 \otimes \ldots \otimes s_n)$ and

$$\begin{array}{rl} X^w = \{\, w' \subseteq w \mid & \text{there is a closed } M : (s \Rightarrow \mathbf{unit}) \\ & \text{such that } [\![M]\!](d){\downarrow} \text{ iff } d \in w' \,\}. \end{array}$$

The path sets in S^w are defined by

$$\begin{array}{c} \{w_1, \ldots, w_k\} \text{ is a path set} \\ \text{if there is an } M : (s \Rightarrow \bar{k}) \text{ such that } [\![M]\!](d) = \mathbf{i} \text{ iff } d \in w_i, \end{array}$$

where

$$\bar{n} = \underbrace{(\mathbf{unit} \oplus \ldots \oplus \mathbf{unit})}_{n}$$

and $\mathbf{1} = inj_1(\top)$, $\mathbf{2} = inj_2(inj_1(\top))$, and so on. It is then not hard to establish the following

Lemma 7 *The set $X = \{ X^w \mid w \in Obj(\mathbf{C}) \}$ defines a $\mathbf{C}$-termination theory T by taking X^w to be a set of models. Moreover, S is a $\mathbf{C}, T$-computational theory.*

The proof uses the alternative characterization of T-models given by Proposition 3.

The next lemma is the main one needed for full abstraction. It proves that every element of the computational relations is represented by a term in Finite FPC.

Lemma 8 *Suppose $w = [s_1, \ldots, s_n]$ and $s = (s_1 \otimes \ldots \otimes s_n)$.*
Then $g \in [\![t]\!](T,S)^w_S$ iff there exists a closed $M : (s \Rightarrow t)$ such that $g = [\![M]\!]$.

The proof proceeds by induction on the structure of t.

5.3 Main results

Theorem 9 (Adequacy) *Suppose M is a closed FPC term of type s. Then*

$$M \Downarrow V \textit{ iff } [\![M]\!]\!\downarrow .$$

Proof: (Sketch) A standard inclusive predicates argument; see [26] for the outline of the proof. ■

Theorem 10 (Full Abstraction) *Suppose M, N are FPC terms. Then*

$$[\![M]\!] \sqsubseteq [\![N]\!] \textit{ iff } M \sqsubseteq_{FPC} N.$$

Proof: (Sketch) The difficult direction to prove is the ($\Leftarrow$) direction. Suppose $M, N : s$ and $[\![M]\!] \not\sqsubseteq [\![N]\!]$. Then there exist closed terms $P^s_n : s \to s^n$, where s^n is a Finite FPC type, such that $[\![P\ M]\!] \not\sqsubseteq [\![P\ N]\!]$ (any **void**'s in the type s^n are replaced by ($\mathbf{rec}\ \alpha.\ \alpha$)). The terms P^s_n are also used in [30]. By

Lemma 8, there exists an $f \in [\![s^n \Rightarrow \mathbf{unit}]\!]$ such that $f([\![P\ M]\!])$ is defined and $f([\![P\ N]\!])$ is not. We know by the concreteness condition that

$$g = (\underline{\lambda} d \in [\,].\ f) \in [\![s^n \Rightarrow \mathbf{unit}]\!](T, S)^{[\,]}.$$

Thus, g is definable by a term $Q : (\mathbf{unit} \Rightarrow s^n \Rightarrow \mathbf{unit})$. Then the context $C[\cdot] = (Q\ \langle\rangle\ (P\ [\cdot]))$ distinguishes M and N, completing the proof. ■

6 Related Work

We have shown how to construct a fully abstract model for call-by-value FPC. It is the first model of a call-by-value language, or one with sums, or one with recursive types to use the logical-relations approach pioneered by Sieber. Our model also supports a simple form of reasoning for showing that certain values are not definable, and yields insight into the structure of sequential computation.

There are other ways to build fully abstract models for FPC. For instance, Riecke and Viswanathan [31, 32] give a dcpo-based model for call-by-value FPC. The construction uses Milner's syntactic methods of [16]. This construction sheds little light into the structure of FPC, except that the model validates least fixpoint reasoning.

Games semantics has also been applied to full abstraction questions for FPC. In [15], McCusker builds a model of *call-by-name* FPC. The sums in this model are *separated*: applying the injection operations to the meaning of a divergent computation returns a convergent value (on which a case expression can branch).

Until recently, it was not known how to adopt games models to the call-by-value setting. Honda and Yoshida have bridged that gap, devising a model for *call-by-value* PCF using games semantics [9]. The model loosens the restrictions of the original games semantics [2, 10, 20] to include strategies that start with the opponent's *answer* rather than a *question*. Intuitively, this means that the value supplied to a call-by-value function is immediately available without interrogation by the player. The basic definitions are quite different from our logical-relations-based model, and the games model sheds light on the process nature of sequentiality.

7 Discussion

Much of the complexity of our model of FPC lies in the use of Kripke relations. On the one hand, since all examples of reasoning in the model seem to require only the "base" relations, it would be interesting to determine when base relations were sufficient. This kind of result might be analogous to Sieber's result that sequentiality relations suffice for proving facts about PCF up to third-order types [33]. On the other hand, recent results of Ralph Loader suggest that one must go beyond base relations to achieve full abstraction. We conjecture that the following problem is undecidable: given a type in Finite FPC, can one decide how many elements there are in the model of that type? If we remained only with the "base" relations, the problem would be *decidable.* The related decision problem for PCF was first pointed out in [12]; see [11, 22] for a further discussion. Loader shows that the decision problem for PCF over the single boolean base type is undecidable [14]. We expect that the proof will carry over to Finite FPC.

We have some hope that the relational account can be adapted to extensions of FPC with other kinds of effects other than simple functional branching, such as continuation-based control operations. We also believe that there is a relationship between "single-threading" of state [21, 23] and sequentiality; it would be interesting to see if our model can be adapted to model a single-threaded global state. One strength of the current model is its clean separation of values and computations. We conjecture that only the definition of "computational relations" must change to reflect the new settings.

Other extensions seem more difficult. For instance, we began by trying to find a similar relation-based model for a *linear* type system, but ran into technical difficulties. Extending FPC with a notion of *local* state, as in Idealized Algol [28] or Standard ML [17], also seems to be difficult. One interesting, though non-trivial, direction would be to extend the language with parametric polymorphism. A different kind of relations would be needed in this instance to model parametricity.

Acknowledgements: The first author thanks the organizers of the Semantics of Computation program at the Newton Institute for Mathematical Sciences, Cambridge, UK where this work was begun. The second author thanks Bell Laboratories for offering a very stimulating and pleasant environment during his visit in the summer of 1996. We also thank Thomas Hildebrandt, Leonid Libkin, Rona Machlin, Peter O'Hearn, Riccardo Pucella, and the anonymous referees for comments.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[2] S. Abramsky, P. Malacaria, and R. Jagadeesan. Full abstraction for PCF (extended abstract). In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, number 789 in Lect. Notes in Computer Sci., pages 1–15. Springer-Verlag, 1994.

[3] E. Astesiano and G. Costa. Nondeterminism and fully abstract models. *RAIRO*, 14(4):323–347, 1980.

[4] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* John Wiley & Sons, 1986.

[5] P.-L. Curien and A. Obtułowicz. Partiality, cartesian closedness, and toposes. *Information and Computation*, 80:50–95, 1989.

[6] P. J. Freyd. Algebraically compact categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini, editors, *Como Category Theory Conference*, volume 1488 of *Lect. Notes in Math.*, pages 95–104. Springer-Verlag, 1991.

[7] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. North-Holland, 1971.

[8] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[9] K. Honda and N. Yoshida. Game theoretic analysis of call-by-value computation. Unpublished manuscript available from "http://hypatia.dcs.qmw.ac.uk", 1997.

[10] J. M. E. Hyland and C.-H. L. Ong. Pi-calculus, dialogue games and PCF. In *7th Annual ACM Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, 1995.

[11] A. Jung, M. Fiore, E. Moggi, P. W. O'Hearn, J. G. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments, and open problems. *Bulletin of the European Association for Theoretical Computer Science*, pages 227–256, June 1996.

[12] A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.*, pages 230–244. Springer-Verlag, 1993.

[13] A. Jung and J. Tiuryn. A new characterization of lambda definability. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.*, pages 245–257. Springer-Verlag, 1993.

[14] R. Loader. Finitary PCF is not decidable. Unpublished manuscript available from "http://hypatia.dcs.qmw.ac.uk", 1996.

[15] G. McCusker. Games and full abstraction for FPC. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 174–183, 1996.

[16] R. Milner. Fully abstract models of the typed lambda calculus. *Theoretical Computer Sci.*, 4:1–22, 1977.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[18] E. Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[19] K. Mulmuley. *Full Abstraction and Semantic Equivalence.* ACM Doctoral Dissertation Award 1986. MIT Press, 1987.

[20] H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, Lect. Notes in Computer Sci. Springer-Verlag, 1994.

[21] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda calculus. Lectures at Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, 1995.

[22] P. W. O'Hearn and J. G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120(1):107–116, 1995.

[23] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42:658–709, 1995.

[24] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[25] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[26] G. D. Plotkin. (Towards a) logic for computable functions. Unpublished manuscript, CSLI Summer School Notes, 1985.

[27] J. C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[28] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[29] J. G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3:387–415, 1993. Preliminary version appears in *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, ACM, 1991.

[30] J. G. Riecke and R. Subrahmanyam. Extensions to type systems can preserve operational equivalences. In M. Hagiya and J. C. Mitchell, editors, *Proceedings of 1994 International Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Lect. Notes in Computer Sci.*, pages 76–95. Springer-Verlag, 1994.

[31] J. G. Riecke and R. Viswanathan. Full abstraction for call-by-value sequential languages. Unpublished manuscript, 1993.

[32] J. G. Riecke and R. Viswanathan. Isolating side effects in sequential languages. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1995.

[33] K. Sieber. Reasoning about sequential functions via logical relations. In *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1992.

[34] A. Stoughton. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Pitman/Wiley, 1988. Revision of Ph.D thesis, Dept. of Computer Science, Univ. Edinburgh, Report No. CST-40-86, 1986.

[35] W. Wechler. *Universal Algebra for Computer Scientists.* Number 25 in EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 1992.