Call-by-Value Games

Samson Abramsky[1] and Guy McCusker[2]

[1] University of Edinburgh, Department of Computer Science, James Clerk Maxwell Building, Edinburgh EH9 3JZ, Scotland. e-mail: samson@dcs.ed.ac.uk
[2] St John's College, Oxford OX1 3JP, England. e-mail: mccusker@comlab.ox.ac.uk

Abstract. A general construction of models of call-by-value from models of call-by-name computation is described. The construction makes essential use of the properties of sum types in common denotational models of call-by-name. When applied to categories of games, it yields fully abstract models of the call-by-value functional language PCF_v, which can be extended to incorporate recursive types, and of a language with local references as in Standard ML.

1 Introduction

In recent years game semantics has emerged as a novel and intuitively appealing approach to modelling programming languages. Its first success was in providing a syntax-free description of a fully abstract model of PCF [10, 1, 15]; full abstraction results have also been obtained for untyped and recursively typed functional languages, as well as languages with imperative features [12, 3]. However, none of this work addressed the problem of modelling *call-by-value* languages—a major shortcoming, given that many real-life languages, including Standard ML, use call-by-value. Recently Honda and Yoshida [9] provided a fully abstract games model for call-by-value PCF (PCF_v), using a new category of games.

In this paper we show that in fact the same framework which provided full abstraction results for call-by-name can also be used to interpret call-by-value, and moreover that full abstraction is retained. The main tool is a general construction which takes a call-by-name model and returns a model of call-by-value computation. It is essentially a matter of programming in a suitable type theory, but we choose to present it at the level of categorical models. We apply our construction to three known models of call-by-name: the usual category of cpos, and two categories of games. In the first case, the familiar model of PCF_v using predomains and partial functions is recovered. In the latter cases, new games models are discovered. The first of these contains a fully abstract model of PCF_v, which can also be extended to provide a fully abstract model of the recursively typed language FPC, while the second is fully abstract for RML, a language with ML-style references.

Our model of PCF_v seems to be an alternative presentation of that of Honda and Yoshida. The results for RML provide the first syntax-free description of a fully abstract model for a language combining dynamically allocated store with higher order procedures. Because the underlying categories of games are not

new, no new proof techniques are required to establish these results. In fact, the proofs of full abstraction can be given at the axiomatic level introduced in [2]. For this reason we concentrate here on describing the general construction and outlining the model definitions, omitting proofs, which can be found elsewhere.

2 The Fam(C) Construction

Call-by-name languages are often viewed as certain extensions of typed λ-calculus. The natural choice of model for call-by-name is therefore a cartesian closed category, equipped with extra structure to account for additional features of the language in question, usually including a fixed point operator to interpret recursion. Such a category should be thought of as a category of domains; the category **Cppo** of complete partial orders with a least element and continuous functions provides a good example. In addition to the products and function spaces which CCCs can interpret, many programming languages also have sum types. However, it is well known that a CCC with fixed points cannot have coproducts, so the interpretation of sums in such a category must be some kind of weak coproduct. For us, a prototypical model of call-by-name is therefore a cartesian closed category with weak coproducts, in a sense to be made precise later.

On the other hand, the work of Moggi [14] shows that call-by-value *can* be interpreted in a CCC with coproducts, that is, a bicartesian closed category, provided there is some extra structure in the shape of a *strong monad*. The leading example of such a model is the category **Cpo** of complete partial orders possibly without a least element, and total continuous functions; the familiar operation of lifting provides a strong monad, and the Kleisli category, in which call-by-value languages are interpreted, is **pCpo**, the category of predomains and partial functions first studied by Plotkin [21].

Let $\mathbf{C}$ be a model of call-by-name computation, that is, a cartesian closed category with weak coproducts; we additionally ask that $\mathbf{C}$ has all small products, not just finite ones. We now construct a bicartesian closed category Fam($\mathbf{C}$) and define a strong monad T on it. The objects of the category Fam($\mathbf{C}$) are families $\{A_i \mid i \in I\}$ of objects of $\mathbf{C}$, indexed by a set I. A map $\phi : \{A_i \mid i \in I\} \to \{B_j \mid j \in J\}$ consists of a ***reindexing function*** $f : I \to J$ and for each $i \in I$, a map $\sigma_i : A_i \to B_{f(i)}$ of $\mathbf{C}$. (This construction is familiar in category theory as the free completion of $\mathbf{C}$ with respect to coproducts.) When $\mathbf{C}$ is **Cppo**, Fam($\mathbf{C}$) can be thought of as the category of disjoint unions of pointed cpos with total continuous functions—a full subcategory of the usual category **Cpo** of predomains.

Lemma 1. Fam($\mathbf{C}$) *is cartesian closed and has coproducts; that is, it is a bicartesian closed category.*

Products are given by

$$\{A_i \mid i \in I\} \times \{B_j \mid j \in J\} \triangleq \{A_i \times B_j \mid (i,j) \in I \times J\};$$

the reindexing function for first projection is itself the first projection $I \times J \to I$, and the map $A_i \times B_j \to A_i$ in each fibre is the projection in $\mathbf{C}$. Similarly, the pairing of two maps consists of the pairing of their reindexing functions and the pairings of the fibrewise maps in $\mathbf{C}$. The terminal object is the singleton family $\{1\}$. The exponential is defined as follows.

$$\{A_i \mid i \in I\} \Rightarrow \{B_j \mid j \in J\} \triangleq \{\Pi_i(A_i \Rightarrow B_{f(i)}) \mid f \in J^I\}$$

(This is why we ask for $\mathbf{C}$ to have all products, not just finite ones.) Given a map $\phi : \{A_i \mid i \in I\} \times \{B_j \mid j \in J\} \to \{C_k \mid k \in K\}$ with reindexing function f and fibrewise maps $\sigma_{i,j}$, the corresponding curried map $\Lambda(\phi)$ is formed by currying f, and taking the map in the ith fibre, which has type

$$A_i \longrightarrow \Pi_j(B_j \Rightarrow C_{f(i,j)})$$

to be the J-fold "pairing" $\langle \Lambda(\sigma_{i,j}) \mid j \in J \rangle$. Coproducts are formed by taking disjoint union of families: given $\{A_i \mid i \in I\}$ and $\{B_j \mid j \in J\}$, their coproduct is $\{C_k \mid k \in I + J\}$, where $C_i = A_i$ for $i \in I$ and $C_j = B_j$ for $j \in J$. The initial object is the empty family $\{\}$.

Thus from any CCC of "domains", we can construct a biCCC of "predomains and total maps". In order to define a suitable monad on $\mathrm{Fam}(\mathbf{C})$, we require $\mathbf{C}$ to have weak coproducts in the following sense. Call some subcollection of the objects of $\mathbf{C}$ ***pointed***, and call some subcategory of the full subcategory of pointed objects the category of ***strict maps***. For any family $\{A_i \mid i \in I\}$ of objects of $\mathbf{C}$, we posit a pointed object $\Sigma_{i \in I} A_i$ and maps $\mathsf{in}_i : A_i \to \Sigma_{i \in I} A_i$ for each i, such that for any family $\sigma_i : A_i \to B$ of maps of $\mathbf{C}$ with B pointed, there is a unique strict map $[\sigma_i \mid i \in I] : \Sigma_{i \in I} A_i \to B$ satisfying $\mathsf{in}_{i_0} ; [\sigma_i \mid i \in I] = \sigma_{i_0}$ for each i_0.

We can now define a monad T on $\mathrm{Fam}(\mathbf{C})$ as follows. Given an object $\{A_i \mid i \in I\}$, define $T\{A_i \mid i \in I\} \triangleq \{\Sigma_{i \in I} A_i\}$, a singleton family. The unit $\eta : \{A_i \mid i \in I\} \to T\{A_i \mid i \in I\}$ has the unique reindexing function, and for each i, the map $\mathsf{in}_i : A_i \to \Sigma_{i \in I} A_i$ of $\mathbf{C}$. The "lifting" operation takes a map $\phi : \{A_i \mid i \in I\} \to T\{B_j \mid j \in J\}$, which must consist of the unique reindexing function and fibrewise maps $\sigma_i : A_i \to \Sigma_{j \in J} B_j$, to the map $\phi^* : T\{A_i \mid i \in I\} \to T\{B_j \mid j \in J\}$, where the morphism in the only fibre is defined to be $[\sigma_i \mid i \in I]$.

Lemma 2. *$(T, \eta, (-)^*)$ is a Kleisli triple; therefore T is a monad on* $\mathrm{Fam}(\mathbf{C})$. *(See [14] for the definitions of these concepts.)*

If $\mathbf{C}$ satisfies some further mild conditions, we can also define a natural transformation $\tau : A \times TB \to T(A \times B)$ making T into a *strong* monad [14]. We will have use for the "double strength" morphism τ_r defined by

$$\begin{array}{ccccc}
TA \times TB & \xrightarrow{\tau} & T(TA \times B) & \xrightarrow{\cong} & T(B \times TA) \\
\downarrow{\scriptstyle \tau_r} & & & & \downarrow{\scriptstyle T\tau} \\
T(A \times B) & \xleftarrow[\cong]{} & T(B \times A) & \xleftarrow[\mathsf{id}^*]{} & T^2(B \times A)
\end{array}$$

and its partner,

$$\tau_l = TA \times TB \xrightarrow{\cong} TB \times TA \xrightarrow{\tau_r} T(B \times A) \xrightarrow{\cong} T(A \times B) \,.$$

These maps do not coincide in general, so T is not necessarily a *commutative* monad [11]. This has certain consequences for us. Given maps $\phi : A \to TB$ and $\psi : A \to TC$ in Fam(**C**), thought of as partial maps from A to B and C respectively, there are two ways of forming a "partial pairing" morphism from A to $T(B \times C)$, namely $\langle f, g \rangle ; \tau_l$ and $\langle f, g \rangle ; \tau_r$. One can think of these as reflecting the distinction between evaluating a pair from left to right and evaluating right to left. In pure functional languages this distinction is redundant; but when we come to consider a language with side-effects, it becomes crucial.

We now have a biCCC equipped with a strong monad T. Furthermore, the Kleisli category for T can be seen as enriched over the original **C**. Therefore, if **C** is a good model of call-by-name computation and has fixed points of all endomorphisms, then all enriched operations on the homsets of the Kleisli category themselves have fixed points. This allows us to interpret recursion in the standard way. One final construction is necessary, that of the ***intrinsic preorder*** on such a category. Define an order on each homset of the biCCC as follows:

$$f \preccurlyeq g : A \to B \iff \forall \alpha : A \Rightarrow B \to T1. \, [`f' ; \alpha = \eta_1 \Rightarrow `g' ; \alpha = \eta_1]$$

where $`f' : 1 \to (A \Rightarrow B)$ is obtained by currying the map f. This induces an equivalence relation on each homset in the usual way. Taking equivalence classes of maps gives another category, which we refer to as the ***extensional quotient*** of Fam(**C**). It is straightforward to show that the biCCC structure and the strong monad lift to this category.

Let us see how all this works in the case of **Cppo**. All the objects are pointed, and strict maps are those which take $\perp$ to $\perp$, as expected. The weak coproduct is given by separated sum. As suggested above, one can think of Fam(**Cppo**) as a full subcategory of the usual category of predomains: each family $\{A_i \mid i \in I\}$ is thought of as the disjoint union of the domains A_i. The monad T takes such a family to the singleton family $\{\Sigma_i A_i\}$, the separated sum of the A_i. Thus T coincides with lifting in this case, so the Kleisli category is a full subcategory of **pCpo**. The intrinsic preorder reduces to the ordinary pointwise ordering on continuous functions in this case, so the extensional quotient is the same as Fam(**Cppo**).

As is well-known, **Cppo** contains a fully abstract model of the language PCF with an additional parallel conditional constant [20]; and **pCpo** contains a fully abstract model of call-by-value PCF with parallel conditional [25]. Therefore our construction takes a category giving the fully abstract model of the call-by-name variant to that of the call-by-value variant. Moreover, when the same construction is applied to the category of games in [12], which contains a fully abstract model of PCF *without* parallel conditional, one obtains a fully abstract model of PCF_v; and when applied to the more liberal category of games in [3], which contains a fully abstract model of Idealized Algol, a fully abstract model for a language with ML-style references is obtained.

3 Games

In this section we give some intuition about how games models work, and explain how to equip the categories introduced in [12] with a weak coproduct, so that the construction described above can be applied. We omit full definitions of the categories involved, which can be found in [12, 3].

A *game* consists of a set of rules by which two participants, Player (P) and Opponent (O) make moves. O moves first, and thereafter the two players alternate. The idea is that O represents the environment or user of a program, while P plays on behalf of the program itself. A simple game is the one used to model the natural numbers in (call-by-name) PCF, of which a typical play is depicted as follows.

$$\begin{array}{c} \mathbf{N} \\ q \\ 3 \end{array}$$

Here O begins by asking the question q, which corresponds to a user "running" a program of type $\mathbb{N}$. P can respond with any number; here he has played the number 3. It is also possible for P to fail to respond; this corresponds to the nontermination of the program. The tensor product of two games is formed by playing them side by side, in interleaved parallel:

$$\begin{array}{cc} \mathbf{N} \otimes \mathbf{N} & \\ q & \\ 3 & \\ & q \\ & 7 \end{array}$$

The linear function space is the same, except that O-moves in the left hand game become P-moves and *vice versa*, so a play might look like this:

$$\begin{array}{cc} \mathbf{N} \multimap \mathbf{N} & \\ & q \\ q & \\ 3 & \\ & 9 \end{array}$$

The play above is a particular run of the "square" program: O begins by demanding output; in order to calculate the output, P first requires input, so demands it using the move q on the left hand side; O supplies the input 3; and finally P can give the output 9. Thus the function $\lambda x.x^2$ becomes a ***strategy*** for P, i.e. a predetermined set of responses to O's moves. Types are therefore modelled as games, and programs are modelled as strategies for those games. The category $\mathcal{G}$ has games as objects, and strategies for the game $A \multimap B$ as morphisms from A to B. Strategies are composed by "parallel composition plus hiding". Let I be the game with no moves (this is the tensor unit). Then the program 3 becomes a strategy for $I \multimap \mathbf{N}$, and composing with the squaring strategy above gives

the following parallel composition:

$$\begin{array}{ccccc} I & \to & \mathbf{N} & \to & \mathbf{N} \\ & & & & q \\ & & q & & \\ & & 3 & & \\ & & & & 9 \end{array}$$

which after hiding the action in the middle game $\mathbf{N}$ is just the strategy for the program $\mathsf{9}$, as expected.

As shown in [12], these ideas can be extended to turn $\mathcal{G}$ into a model of linear logic [8]. The subcategory $\mathcal{G}_{\mathrm{inn}}$ has as morphisms not all strategies but just the ***innocent*** ones. (A strategy is innocent if it bases its decision of what move to make not on the entire history of play so far but only on a certain subsequence of it, called the ***view***.) This too is a model of linear logic.

The categories $\mathcal{G}$ and $\mathcal{G}_{\mathrm{inn}}$ give rise to CCCs $\mathcal{I}$ and $\mathcal{I}_{\mathrm{inn}}$, the so called ***intensional*** categories of [12, 3], in which maps are strategies for $!A \multimap B$ rather than simply $A \multimap B$. The game $!A$ is a version of A in which moves can be repeated, so that a single play of $!A$ consists of several interleaved plays of A. By applying the construction of the previous section to these CCCs, we obtain categories $\mathrm{Fam}(\mathcal{I})$ and $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$, in which we can model call-by-value computation.

Some words of warning The sketch above is incomplete. The formal definition of game introduces the notion of *justification*: when a move is played, a pointer is attached to it indicating which earlier move justifies it. This facilitates the definition of *view* of a sequence, and hence of innocent strategies and the categories $\mathcal{G}_{\mathrm{inn}}$ and $\mathcal{I}_{\mathrm{inn}}$. Details of these definitions, which build on the pioneering work of Hyland, Ong and Nickau [10, 15], can be found in [12, 3].

3.1 Weak Coproducts

The final piece of structure we need to apply our construction to the games models is a weak coproduct in $\mathcal{I}$ and $\mathcal{I}_{\mathrm{inn}}$. In fact we shall obtain this via a weak coproduct on $\mathcal{G}$ and $\mathcal{G}_{\mathrm{inn}}$, which we now define. Say that a game is ***pointed*** if it is well-opened (a technical condition; see [12]) and has a unique initial move q. A map $\sigma : A \to B$ between pointed games is ***strict*** if it responds to the initial move in B with the initial move in A. For any family of games $\{A_i \mid i \in I\}$, the pointed game $\Sigma_{i \in I} A_i$ is defined as follows. Its set of moves is the disjoint union of the moves of the A_i, together with fresh moves q and $\{i \mid i \in I\}$. The move q is the unique initial move, to which P can respond with any of the i as an answer. After i, play continues as in A_i, except that initial moves of A_i are now justified by i.

The strategy $\mathsf{in}_{i_0} : A_{i_0} \to \Sigma_{i \in I} A_i$ responds to the initial question with i_0 and thereafter plays copycat between the two available copies of A_{i_0}.

Given a family of maps $\sigma_i : A_i \to B$ with B pointed, the strict strategy $[\sigma_i \mid i \in I] : \Sigma_{i \in I} A_i \multimap B$ plays as follows. It responds to the initial move of

B with that of $\Sigma_{i\in I} A_i$, and after O plays a move i, it continues playing as σ_i would play. It is clear that composing this with in_{i_0} yields σ_{i_0} for any $i_0 \in I$; furthermore it is the unique strict strategy with this property.

This construction lifts to $\mathcal{I}$ and $\mathcal{I}_{\mathrm{inn}}$ as follows. The pointed games are as above, but strict maps are now those of the form $\mathsf{der}\,;\sigma : !A \multimap B$ where $\sigma : A \multimap B$ is a strict map in $\mathcal{G}$ or $\mathcal{G}_{\mathrm{inn}}$. The sum of a family $\{A_i \mid i \in I\}$ is given by $\{\Sigma_i !A_i\}$, with injections given by the strategies $\mathsf{in}_i : !A_i \multimap \Sigma_i !A_i$ as above. Given a family of maps $\sigma_i : A_i \to B$ with B pointed, that is strategies $\sigma_i : !A_i \multimap B$, the copairing is given by the strategy $\mathsf{der}\,;[\sigma_i \mid i \in I] : !\Sigma_i !A_i \multimap B$. This corresponds precisely to Girard's translation of intuitionistic disjunction into linear logic. It is worth noting that the interpretation of sum types used in [13] did not take this form but rather used the linear sum $\Sigma_i A_i$ in $\mathcal{I}$ as well as in $\mathcal{G}$; this led to a rather unsatisfactory situation in which good models of sum types only existed after taking the extensional quotient. Moving to Girard's translation rectifies this, and furthermore gives the same interpretation of sums after extensional quotient.

3.2 Applying the Construction

Let us see what happens when we apply our construction to $\mathcal{I}$ to obtain a games model of call-by-value. $\mathrm{Fam}(\mathcal{I})$ has families of games as objects; a map from $\{A_i \mid i \in I\}$ to $\{B_j \mid j \in J\}$ consists of a function $f : I \to J$ and for each $i \in I$, a strategy $\sigma_i : !A_i \multimap B_j$. Recalling the definition of the monad T, a map from $\{A_i \mid i \in I\}$ to $T\{B_j \mid j \in J\}$ is a family of strategies

$$\sigma_i : !A_i \multimap \Sigma_{j\in J} !B_j.$$

By the universal property of the weak coproduct, such a family can alternatively be seen as a single strict strategy

$$\sigma : \Sigma_{i\in I} !A_i \multimap \Sigma_{j\in J} !B_j$$

Thus the Kleisli category for T can be seen as a subcategory of $\mathcal{G}$ consisting of games of the form $\Sigma_{i\in I} !A_i$, and strict strategies.

When modelling programming languages, the type of natural numbers is interpreted as the family $\{1_n \mid n \in \mathbb{N}\}$. Under the above representation, this is the game $\Sigma_{n\in\mathbb{N}} 1$, which is precisely the game $\mathbf{N}$ described previously. The type of pairs of naturals is $\{1_{(n_1,n_2)} \mid (n_1, n_2) \in \mathbb{N} \times \mathbb{N}\}$. As a game, it begins with a unique initial question by O, to which P responds by giving a *pair* of natural numbers. This should be contrasted with the call-by-name interpretation in which O can begin by asking a question either about the first component of the pair or about the second, to which P responds with a single natural number in either case. This distinction should be seen as an intensional description of the difference between product and smash product of cpos in domain models.

3.3 Relationship with Honda-Yoshida

Under the representation above, the initial questions of games are somewhat redundant: every play of every strategy

$$\sigma : \Sigma_{i \in I} !A_i \multimap \Sigma_{j \in J} !B_j$$

in the model begins with a unique initial question on the right hand side, to which the strategy responds by asking the unique initial question of the left hand side. It is possible to reformulate the model so that these moves are elided and play begins with O supplying an index i on the left. This economical representation more closely resembles the model of call-by-value proposed by Honda and Yoshida [9]. We believe their model of PCF_v to be the same as ours but presented in this way.

4 PCF_v

Perhaps the simplest typed call-by-value functional programming language is PCF_v, a simply-typed λ-calculus with constants for arithmetic and recursion. Its types are given by the grammar

$$A ::= \mathtt{exp} \mid A \to A.$$

The canonical forms of the language are

$$c ::= x \mid \mathtt{n} \mid \mathtt{succ} \mid \mathtt{pred} \mid \mathtt{Y}_{A_1,A_2} \mid \lambda x.M$$

where n ranges over the natural numbers, x over a countable collection of variables, and M over the general terms, which are defined by

$$M ::= c \mid MM \mid \mathtt{if0}_A M\ M\ M.$$

The constants $\mathtt{n}$ have type $\mathtt{exp}$; $\mathtt{succ}$ and $\mathtt{pred}$ have type $\mathtt{exp} \to \mathtt{exp}$; for types A_1 and A_2, $\mathtt{Y}_{A_1,A_2}$ has type $((A_1 \to A_2) \to (A_1 \to A_2)) \to (A_1 \to A_2)$; and given terms M of type $\mathtt{exp}$ and N_1, N_2 of type A the term $\mathtt{if0}_A M\ N_1\ N_2$ has type A. We omit the standard definition of terms-in-context

$$\Gamma \vdash M : A$$

in which $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a list of distinct variables, tagged with types, containing all the variables free in M. The operational semantics is defined in Figure 1.

The relation of ***observation equivalence*** between terms-in-context $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$ is defined as follows. We write $\Gamma \vdash M \sim N$, or just $M \sim N$, iff for all contexts $C[-]$ such that $C[M]$ and $C[N]$ are closed terms, $C[M]{\Downarrow} \iff C[N]{\Downarrow}$. (See [19] for a more detailed definition of contexts and substitution of terms for "holes".)

Functions

$$\frac{}{c \Downarrow c} \qquad \frac{M \Downarrow \lambda x.M' \quad N \Downarrow N' \quad M'[N'/x] \Downarrow P}{MN \Downarrow P}$$

Arithmetic

$$\frac{M \Downarrow \mathtt{succ} \quad N \Downarrow \mathtt{n}}{MN \Downarrow \mathtt{n+1}} \qquad \frac{M \Downarrow \mathtt{pred} \quad N \Downarrow \mathtt{n+1}}{MN \Downarrow \mathtt{n}} \qquad \frac{M \Downarrow \mathtt{pred} \quad N \Downarrow \mathtt{0}}{MN \Downarrow \mathtt{0}}$$

Conditionals

$$\frac{M \Downarrow \mathtt{0} \quad N_1 \Downarrow V}{\mathtt{if0}\ M\ N_1\ N_2 \Downarrow V} \qquad \frac{M \Downarrow \mathtt{n+1} \quad N_2 \Downarrow V}{\mathtt{if0}\ M\ N_1\ N_2 \Downarrow V}$$

Recursion

$$\frac{M \Downarrow \mathtt{Y} \quad N \Downarrow V}{MN \Downarrow \lambda x.V(YV)x}$$

Fig. 1. Operational semantics of PCF_v

The interpretation of PCF_v in our categories $\mathrm{Fam}(\mathcal{I})$ and $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ of games is standard: the type `exp` is interpreted as the object $\mathbf{N} = \{1_n \mid n \in \mathbb{N}\}$, which is the coproduct of countably many copies of the terminal object, and each type $A \to B$ is interpreted as $[\![A]\!] \Rightarrow T[\![B]\!]$. A term-in-context $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ is interpreted as a morphism

$$[\![x_1 : A_1, \ldots, x_n : A_n \vdash M : B]\!] : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to T[\![B]\!]$$

as follows. Each numeral $\mathtt{n}$ can be interpreted as $n; \eta : 1 \to T\mathbf{N}$, where n is the nth coproduct insertion. The constant `succ` is interpreted using the coproduct structure of $\mathbf{N}$ as the map

$$[[\mathsf{1}], [\mathsf{2}], \ldots] : \mathbf{N} \to T\mathbf{N};$$

`pred` and `if0` are handled similarly. To model `Y` we make use of the enrichment of the categories $\mathrm{Fam}(\mathcal{I})$ and $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ over $\mathcal{I}$ and $\mathcal{I}_{\mathrm{inn}}$ respectively to obtain a fixed point combinator. In fact we can also consider these categories as being **Cppo** enriched and use the usual order-theoretic construction of a least fixed point operator. The following standard definitions complete the semantics.

$$\begin{aligned}
[\![\Gamma, x : A \vdash x : A]\!] &= \pi_2 \,;\, \eta_A : [\![\Gamma]\!] \times [\![A]\!] \to T[\![A]\!] \\
[\![\Gamma \vdash \lambda x.M : A \to B]\!] &= \Lambda([\![\Gamma, x : A \vdash M : B]\!]) \,;\, \eta_{[\![A]\!] \Rightarrow T[\![B]\!]} : [\![\Gamma]\!] \to T([\![A]\!] \Rightarrow T[\![B]\!]) \\
[\![\Gamma \vdash MN : B]\!] &= \langle [\![M]\!], [\![N]\!] \rangle \,;\, \tau_l \,;\, \mathsf{ev}^* : [\![\Gamma]\!] \to T[\![B]\!]
\end{aligned}$$

Notice that in the semantics of application MN, we use τ_l rather than τ_r to reflect the fact that our intended interpreter evaluates M before N.

Proposition 3. *For closed terms M, if $M \Downarrow V$ then $[\![M]\!] = [\![V]\!] \neq \bot$.*

Proposition 4. *For closed terms M, if $[\![M]\!] \neq \bot$ then $M\Downarrow$.*

The first of these is proved by straightforward induction. The second requires either a computability predicate argument as in [20] or the use of a logical relation as in [21]; both methods are completely standard. An easy consequence of these results is:

Proposition 5 (Soundness). *For any $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, if $[\![M]\!] = [\![N]\!]$ then $M \sim N$.*

It should be emphasized that these results hold for both the innocent and knowing games models. However, the following definability result holds *only* for innocent strategies. Its proof is a mild generalization of that for PCF [1, 2].

Proposition 6 (Innocent Definability). *Let $A_1, \ldots, A_n$ and B be types of PCF_v, and let $\phi : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to T[\![B]\!]$ be any compact morphism in $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$. Then there exists a term $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ of PCF_v such that $[\![M]\!] = \phi$.*

It is now easy to show that the model of PCF_v provided by $\mathcal{E}_{\mathrm{inn}}$, the extensional quotient of $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$, is fully abstract.

Theorem 7 (Full Abstraction for PCF_v). *The model of PCF_v in $\mathcal{E}_{\mathrm{inn}}$ is fully abstract.*

Proof. Soundness follows easily from the soundness of the model in $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$. For completeness, suppose that $[\![M]\!] \neq [\![N]\!]$ for some (wlog closed) terms M and N of type A. By definition of $\mathcal{E}_{\mathrm{inn}}$, there exists a compact morphism $\alpha : TA \to T1$ such that (wlog) $[\![M]\!]\,;\alpha = \bot$ and $[\![N]\!]\,;\alpha \neq \bot$. In fact we can use this to find a compact morphism $\beta : (\mathbf{N} \Rightarrow T\mathbf{N}) \to T\mathbf{N}$ such that $[\![\lambda x.M]\!]\,;\beta = \bot$ while $[\![\lambda x.N]\!]\,;\beta \neq \bot$. By Innocent Definability, $\beta = [\![y \vdash C[y]]\!]$ for some term $C[y]$, and then we have $[\![C[\lambda x.M]]\!] = \bot$ and $[\![C[\lambda x.N]]\!] \neq \bot$. Therefore $C[\lambda x.M]\Uparrow$ while $C[\lambda x.N]\Downarrow$ so $M \not\sim N$.

5 Recursive Types

Since our categories have products and coproducts, it is a straightforward matter to extend the above results to a version of PCF_v augmented with product and sum types. Another natural (and very useful) extension is to add *recursive types*; once this has been done, the type of natural numbers with its associated constants and the $\mathtt{Y}$ combinator become *definable*, so one is led to the very simple syntax of Plotkin's language FPC [21]. The types are

$$\tau ::= \mathtt{T} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \mu\mathtt{T}.\tau$$

where $\mathtt{T}$ ranges over a countable collection of type variables.

The standard way to interpret such a type system in a category $\mathbf{C}$ is to model a type τ with a free type variable $\mathtt{T}$ as a functor

$$[\![\tau]\!] : \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{C}.$$

(The mixed variance of this functor is to account for such types as $\mathtt{T} \to \mathtt{T}$—the positive and negative occurrences of $\mathtt{T}$ are treated separately.) The closed type $\mu\mathtt{T}.\tau$ is then interpreted as an object D of $\mathbf{C}$ satisfying $D \cong [\![\tau]\!](D, D)$, along with some other conditions specifying that it is the *canonical solution* (this is Freyd's notion of *minimal invariant* [7, 6, 16]).

However, while the operations corresponding to sums and partial function spaces are functorial on the Kleisli category for the monad T, the lack of commutativity of T means that the product operation does not necessarily lift to a bifunctor on the Kleisli category: it gives rise to a premonoidal structure [22] rather than a monoidal one. This fact has already manifested itself harmlessly in the existence of two different pairing operators; we now find that it forms an obstacle to interpreting recursive types.

All is not lost! Both the Kleisli categories of $\mathrm{Fam}(\mathcal{I})$ and $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ have a subcategory containing all their objects, on which all of the type constructors are functorial. A map in either of these Kleisli categories consists of a family of strategies

$$\sigma_i : !A_i \multimap \Sigma_{j \in J} !B_j.$$

Such a family is included in the corresponding subcategory iff each σ_i is either the empty strategy $\bot$ or factors as $\tau_i^\dagger ; \mathsf{in}_j$ for some $j \in J$ and $\tau : !A_i \to B_j$. (This subcategory is equivalent to the Kleisli category on $\mathrm{Fam}(\mathcal{I})$ or $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ for the monad $(-) + 1$.)

It is a simple matter to check that the operations of product, sum and function space are all functorial on this subcategory. The results of [12] can then be adapted to show that these functors have minimal invariants, so all the closed types of FPC have an interpretation as objects of these subcategories, and hence of the whole Kleisli categories. Furthermore, as in *loc. cit.*, Pitts' theory of *invariant relations* can be used to show that the model of FPC thus obtained is sound. The analogue of the above Innocent Definability result also holds for this model, so we can establish the following.

Theorem 8 (Full Abstraction for FPC). *Both* $\mathrm{Fam}(\mathcal{I})$ *and* $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$*, and therefore also their extensional quotients* $\mathcal{E}$ *and* $\mathcal{E}_{\mathrm{inn}}$*, contain sound models of* FPC*. Every compact morphism of finite* FPC*-type in* $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ *is definable, and hence the model in* $\mathcal{E}_{\mathrm{inn}}$ *is fully abstract.*

Notice that these results are for a version of FPC with *call-by-value* operational semantics, as in [5, 4], rather than the call-by-name variant for which a fully abstract games model was provided in [13, 12]. Note also that, thanks to an improved interpretation of sums in $\mathcal{I}$ and $\mathcal{I}_{\mathrm{inn}}$, the model, along with the soundness and definability results, is given in the intensional categories, while previously models existed only at the extensional level.

6 A Language with Store

A more radical extension to PCF_v is the addition of a mechanism for the generation of program variables. The expressive power of such an extended language is very great: for example, objects and classes, at least in a simple form, can then be considered as syntactic sugar [23]. However, the price for this extra power is that the behaviour of programs becomes quite subtle and difficult to reason about.

There are two distinct ways in which such variables can be allocated. The first, exemplified by Algol-like languages [24], employs a *block structure*, in which variables local to blocks are allocated on entry to the block and deallocated on exit. The second method, used in languages such as Standard ML, is referred to as *dynamic allocation*. Here, it is possible for access to a variable allocated in a block to be passed outside that block, so a variable, once allocated, must be seen as persisting forever. A detailed analysis of the behaviour of languages with this feature has been undertaken by Pitts and Stark [26, 18, 17] using both operational and denotational techniques, but until now, no fully abstract model was known.

A fully abstract games model has already been given for Idealized Algol using the category $\mathcal{I}$ [3]. We now show that the category $\mathrm{Fam}(\mathcal{I})$ is fully abstract for a language with dynamically allocated references. It may seem surprising that modifying our game semantics for Idealized Algol, which has stack-based storage, by applying such a general construction gives rise to a model of a language with dynamically allocated storage and "scope extrusion" effects. In fact, this illustrates the considerable flexibility afforded by the underlying Linear types. The model for Idealized Algol uses the standard co-Kleisli construction whereby morphisms are linear arrows of the form

$$f : !A \to B$$

with composition using "promotion" $f^\dagger$. This means that, even though the strategy f may be "history-sensitive", it has to start again from scratch on each call, and persistent storage effects will not be modelled. By contrast, the underlying Linear arrows in the $\mathrm{Fam}(\mathcal{I})$ model are (simplifying slightly) of the form

$$f : !A \to !B$$

and composition is simply composition from the Linear category. This means that history-sensitive strategies *can* carry information over from one invocation to the next, and the effects characteristic of heap-allocated storage and scope extrusion can indeed be captured.

We now describe the language of study. It bears a close resemblance to Reduced ML as studied by Stark, with one important distinction, discussed below. We therefore call our language RML.

Let us first add to PCF_v a type `com` of commands akin to the `unit` type of Standard ML: a command is viewed as a "function" which has side-effects and then returns a trivial value. The command with no side-effect is a canonical

form `skip`. Let us also add a type `var` of variables which store natural numbers, together with constants

$$\begin{aligned} \texttt{deref} &: \texttt{var} \to \texttt{com} \\ \texttt{assign} &: \texttt{var} \to \texttt{exp} \to \texttt{com} \\ \texttt{new} &: \texttt{var} \end{aligned}$$

for dereferencing, assignment and allocation of fresh variables. With these constants a large variety of imperative constructs can be defined as abbreviations. For instance,

$$\begin{aligned} \texttt{new}\ x\ \texttt{in}\ C\ \texttt{end} &\triangleq (\lambda x : \texttt{var}.\, C)\texttt{new} \\ x := 1; M &\triangleq (\lambda d : \texttt{com}.\, M)(\texttt{assign}\ x\ 1) \qquad d \text{ not in } M. \end{aligned}$$

Notice that call-by-value is essential here.

Following Reynolds' analysis of variables in Algol-like languages, we identify the type `var` with the product of its two "access methods": dereferencing, which produces a (computation of a) natural number, and assignment, which takes an expression and produces a command:

$$[\![\texttt{var}]\!] = T[\![\texttt{exp}]\!] \times ([\![\texttt{exp}]\!] \Rightarrow T[\![\texttt{com}]\!]).$$

To make this identification complete, we add to the language a constant for creating new "variable objects"

$$\texttt{mkvar} : (\texttt{com} \to \texttt{exp}) \to (\texttt{exp} \to \texttt{com}) \to \texttt{var}.$$

Semantically, `mkvar` is essentially the pairing constructor for the `var` type. The introduction of such a constant means that so-called "bad variables", i.e. terms of type `var` which do not denote actual memory locations, play an important role in the language. In particular, it means that RML is *not* a conservative extension of Stark's Reduced ML. It also implies that an equality test for the `var` type would be meaningless: in general it makes no sense to ask whether two terms of type `var` denote the same memory location. It is worth mentioning that, conversely, in languages in which all variables are "good", equality of references is *definable*. (The idea is simply to write different values into the two variables, and then test if they contain the same value.) Thus there are two key differences between the references of our language and those of Standard ML: first, RML only allows storage of values of ground type, while Standard ML references can store any type; second, bad variables exist.

The absence of equality of references from our language (or, equivalently, the presence of bad variables) may be regarded as a greater or lesser defect depending on one's view of a number of related issues. For example, in a language such as ML where references may store values of any type, `var` becomes a type constructor $\texttt{var}[X]$. Peter O'Hearn has pointed out that in the absence of bad variables, this type constructor is not functorial, and in fact does not even preserve isomorphisms! This raises serious questions about the denotation

Variables

$$\frac{M_1 \Downarrow V_1 \quad M_2 \Downarrow V_2}{\texttt{mkvar}\ M\ N \Downarrow \texttt{mkvar}\ V_1\ V_2} \qquad \frac{}{\langle L, s\rangle \texttt{new} \Downarrow \langle L \cup \{l\}, s\rangle l}\ l \notin L$$

Assignment

$$\frac{\langle L, s\rangle M \Downarrow \langle L', s'\rangle l \quad \langle L', s'\rangle N \Downarrow \langle L'', s''\rangle \texttt{n}}{\langle L, s\rangle \texttt{assign}\ M\ N \Downarrow \langle L'', s''(l \mapsto n)\rangle \texttt{skip}}$$

$$\frac{M \Downarrow \texttt{mkvar}\ V_1\ V_2 \quad N \Downarrow \texttt{n} \quad V_2(\texttt{n}) \Downarrow \texttt{skip}}{\texttt{assign}\ M\ N \Downarrow \texttt{skip}}$$

Dereferencing

$$\frac{\langle L, s\rangle M \Downarrow \langle L', s'\rangle l \quad s'(l) = n}{\langle L, s\rangle \texttt{deref}\ M \Downarrow \langle L', s'\rangle \texttt{n}} \qquad \frac{M \Downarrow \texttt{mkvar}\ V_1\ V_2 \quad V_1(\texttt{skip}) \Downarrow \texttt{n}}{\texttt{deref}\ M \Downarrow \texttt{n}}$$

Fig. 2. Operational semantics of RML

of recursive types in such languages. At any rate, finding a fully abstract model for a language with equality of references must be left as a challenge for future work.

The operational semantics is given in terms of stores as follows. Let l range over a countable collection of ***locations***, and L over finite sets of locations. An L-store s is a partial function from L to natural numbers. We write $s(l \mapsto n)$ for the store obtained by updating s so that l is mapped to n; this operation may extend the domain of s. The operational semantics takes the form of judgements

$$\langle L, s\rangle M \Downarrow \langle L', s'\rangle N$$

where s is an L-store and s' is an L'-store, and $L \subseteq L'$. Adopting the convention that a rule

$$\frac{M_1 \Downarrow V_1 \cdots M_n \Downarrow V_n}{M \Downarrow V}$$

is an abbreviation for

$$\frac{\langle L_1, s_1\rangle M_1 \Downarrow \langle L_2, s_2\rangle V_1, \langle L_2, s_2\rangle M_2 \Downarrow \langle L_3, s_3\rangle V_2, \ldots \langle L_n, s_n\rangle M_n \Downarrow \langle L_{n+1}, s_{n+1}\rangle V_n}{\langle L_1, s_1\rangle M \Downarrow \langle L_{n+1}, s_{n+1}\rangle V}$$

the only additions to the operational semantics are as shown in Figure 2.

Notice that locations l appear in the operational semantics as syntax; they are nonetheless not part of the "official" syntax of the language, and do not appear in user programs. All access to locations must be allocated using `new`.

Let us outline the interpretation of RML in $\mathrm{Fam}(\mathcal{I})$, as an extension of that of PCF_v. The type com is interpreted as the terminal object, and the interpretation of var is as above. We can then define

$$\begin{aligned}
[\![\mathtt{deref}]\!] &= \pi_1 : T\mathbf{N} \times (\mathbf{N} \to T1) \to T\mathbf{N} \\
[\![\mathtt{assign}]\!] &= \pi_2 \,;\, \eta : T\mathbf{N} \times (\mathbf{N} \to T1) \to T(\mathbf{N} \to T1) \\
[\![\mathtt{skip}]\!] &= \eta_1 : 1 \to T1.
\end{aligned}$$

The interpretation of mkvar is a suitable currying of the pairing constructor. Note that all the above maps exist in $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ as well as $\mathrm{Fam}(\mathcal{I})$, so we can interpret the whole of RML *except* the new constant using innocent strategies. We call this sublanguage $\mathsf{RML} - \{\mathtt{new}\}$.

To interpret new, we give a morphism $\mathsf{cell} : 1 \to T[\![\mathtt{var}]\!]$, that is, a strategy for the game $\Sigma_*!(\Sigma_n!1 \times (\Sigma_*1)^{\mathbb{N}})$. Here Σ_* means sum over a singleton set, whose unique element we call $*$. The game Σ_*1 has plays of the form $q \cdot *$, so $(\Sigma_*1)^{\mathbb{N}}$ consists of countably many disjoint copies of this game. We write the move q from the nth copy as $\mathsf{write}(n)$, and the move $*$ in each copy as ok. Similarly, the play $q \cdot n$ in $\Sigma_n!1$ is written as $\mathsf{read} \cdot n$. Thus plays of $T[\![\mathtt{var}]\!]$ look like this:

$$q \cdot * \cdot \mathsf{write}(3) \cdot \mathsf{ok} \cdot \mathsf{read} \cdot 17 \cdots$$

The strategy cell is the obvious one: it responds to the initial q with $*$, to indicate convergence. Thereafter, it responds to each $\mathsf{write}(n)$ with ok, and to each read with the last value written, if any. This strategy is *not* innocent, so we do not model all of RML in $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$: the extra freedom of using arbitrary strategies is necessary here.

Proposition 9 (Soundness). *The model of* RML *in* $\mathrm{Fam}(\mathcal{I})$*, and hence in* $\mathcal{E}$*, is sound, i.e. if* $[\![M]\!] = [\![N]\!]$ *then* $M \sim N$.

This is proved in a similar way to the soundness result for Idealized Algol [3].

For $\mathsf{RML} - \{\mathtt{new}\}$, which can be modelled using innocent strategies, a definability result is available, and is proved just as for PCF_v.

Proposition 10 (Innocent Definability for RML). *Every compact morphism of* RML*-type in* $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ *is definable by some term of* $\mathsf{RML} - \{\mathtt{new}\}$.

We also have the following result from [3]:

Proposition 11 (Innocent Factorization). *Let* A *be a type of* RML *and* $\phi : 1 \to T[\![A]\!]$ *a morphism in* $\mathrm{Fam}(\mathcal{I})$*. Then there exists* $\psi : [\![\mathtt{var}]\!] \to T[\![A]\!]$ *in* $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$ *such that* $\mathsf{cell} \,;\, \psi^* = \phi$*. Furthermore, if* ϕ *is compact in* $\mathrm{Fam}(\mathcal{I})$*,* ψ *can be chosen compact in* $\mathrm{Fam}(\mathcal{I}_{\mathrm{inn}})$.

The idea behind the proof of this result is that the innocent strategy ψ simulates the knowing strategy ϕ by using the storage cell to record the history of play. The factorization gives rise to:

Proposition 12 (Definability for RML). *Every compact morphism of* RML *type in* $\mathrm{Fam}(\mathcal{I})$ *is definable by a term of* RML.

Proof. We just consider the case of a morphism $\phi : 1 \to T[\![A]\!]$. By the Proposition above, this factorizes as $\mathsf{cell}; \psi^*$ for some compact innocent map $\psi : [\![\mathtt{var}]\!] \to T[\![A]\!]$. By Innocent Definability, $\psi = [\![x : \mathtt{var} \vdash M : A]\!]$ for some M, and hence $\phi = [\![(\lambda x.M)\mathtt{new}]\!]$.

Finally, just as for PCF_v, the Definability result allows us to prove full abstraction.

Theorem 13 (Full Abstraction for RML). *The model of* RML *in $\mathcal{E}$ is fully abstract.*

References

1. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. Accepted for publication in Information and Computation, 1997.
2. S. Abramsky. Axioms for full abstraction and full completeness. In G. Plotkin and C. Sterling, editors, *Milner Festschrift*. MIT Press, to appear.
3. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-like Languages*, pages 297–329 of volume 2. Birkhaüser, 1997.
4. M. P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.
5. M. P. Fiore and G. D. Plotkin. An axiomatization of computationally adequate domain theoretic models of FPC. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 92–102. IEEE Computer Society Press, 1994.
6. P. J. Freyd. Recursive types reduced to inductive types. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1990.
7. P. J. Freyd. Algebraically complete categories. In A. Carboni et al., editors, *Proc. 1990 Como Category Theory Conference*, pages 95–104, Berlin, 1991. Springer-Verlag. Lecture Notes in Mathematics Vol. 1488.
8. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
9. K. Honda and N. Yoshida. Game theoretic analysis of call-by-value computation. In P. Degano, R. Gorrieri, and A. Marchietti-Spaccamela, editors, *Proceedings, 25th International Colloquium on Automata, Languages and Programming: ICALP '97*, volume 1256 of *Lecture Notes in Computer Science*, pages 225–236. Springer-Verlag, 1997.
10. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. Accepted for publication in Information and Computation, 1997.
11. B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.
12. G. McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. PhD thesis, Department of Computing, Imperial College, University of London, 1996.
13. G. McCusker. Games and full abstraction for FPC. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 174–183. IEEE Computer Society Press, 1996.

14. E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
15. H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, Lecture notes in Computer Science. Springer, 1994.
16. A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
17. A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.
18. A. M. Pitts and I. D. B. Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *Workshop on State in Programming Languages, Copenhagen, 1993*, pages 31–45. ACM SIGPLAN, 1993. Yale Univ. Dept. Computer Science Technical Report YALEU/DCS/RR-968.
19. A. M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus, December 1994. vi+135 pp, draft version.
20. G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
21. G. Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.
22. J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 11, 1993.
23. J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.
24. J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.
25. K. Sieber. Relating full abstraction results for different programming languages. In *Proceedings of 10th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore*, 1990. Springer LNCS 472.
26. I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Dec 1994.