A fully abstract game semantics for general references

Samson Abramsky Kohei Honda
LFCS, University of Edinburgh
{samson,kohei}@dcs.ed.ac.uk

Guy McCusker
St John's College, Oxford
mccusker@comlab.ox.ac.uk

Abstract

A games model of a programming language with higher-order store in the style of ML-references is introduced. The category used for the model is obtained by relaxing certain behavioural conditions on a category of games previously used to provide fully abstract models of pure functional languages. The model is shown to be fully abstract by means of factorization arguments which reduce the question of definability for the language with higher-order store to that for its purely functional fragment.

1 Introduction

Over the past few years, game semantics has been used to give the first syntax-independent constructions of fully abstract models for a number of programming languages, including the prototypical functional language PCF [1, 8, 20], functional languages with richer type structures or different evaluation strategies [2, 3, 7, 12, 13], and languages with imperative features such as first-order references and non-local control constructs [4, 5, 11]. In the present paper, we obtain the first full abstraction result for a language with *general references*.

By general references we mean references which can store not only values of ground types (integers, booleans, etc.) but also those of higher types (procedures, higher-order functions, or references themselves). The general reference is a useful and powerful programming construct: the ability to store and pass arbitrary references is essential not only for efficiency but also for clarity of program structures involving stateful entities, as in object-oriented programming. Indeed, the behavioural aspects of object-oriented programming can be cleanly and directly represented once one has general references (cf. Section 2.3). The power of general references in typed imperative languages may also be seen from the observation that a basic feature of "mobility" or "scope extrusion" in the π-calculus [17], namely the ability to pass references to arbitrary behaviours out of the scope in which they were created, is already present in this setting. This feature is essential in object-oriented programming where one often needs to pass a reference to a newly created object to another object.

The key idea behind our model is to represent a reference by a certain form of *information flow*: a reference is modelled not as a static entity, but as a dynamic behaviour which mediates the flow of information between readers and writers, connecting them in an appropriate fashion. In the presence of higher-order references, these connections have to be made dynamically, and the computations of the multiple readers and writers may be interleaved in arbitrarily complex ways. The technical apparatus of game semantics provides exactly the right setting in which to formalize this idea. The "dynamic connections" used to interpret references are modelled concretely by *copy-cat behaviour*, a ubiquitous notion in game semantics; and the categorical structure provided by games allows this modelling of references to be integrated smoothly and compositionally with the interpretation of the other program phrases. This dynamic representation of references is quite different in character to the traditional location-based models of store [15, 21, 25] (or related denotational models of π-calculus [6, 24]). It forms the basis of a precise semantic account of programs with general references, as shown by the full abstraction result.

Technically, our results follow a similar pattern to those in [4, 5, 11], and exemplify the paradigm of the "semantic cube" proposed by the first author. In this paradigm, we consider firstly a "syntactic cube" of extended typed λ-calculi. The "origin" of this cube is occupied by a purely functional language. Each "axis" of the cube corresponds to the extension of the purely functional language by some non-functional feature, such as state or control. Corresponding to this syntactic cube, there is a "semantic cube" of various categories of games and strategies. The origin of the semantic cube is occupied by the category of highly constrained strategies which correspond to the discipline of purely functional programming. These constraints include "innocence" and "well-bracketing" as in [8, 20]. Each axis of the semantic cube corresponds to the *relaxation* of one of these semantic constraints on strategies. Thus we get categories of innocent but non-well-bracketed strategies [11], of well-bracketed but "knowing" (i.e. non-innocent) strategies [4, 5], etc. Remarkably, there is a precise correspondence between these syntactic and semantic cubes. Thus for example relaxing the constraint of innocence allows us to model first-order references [4], while relaxing the well-bracketing constraint allows control to be modelled [11]. Moreover, these increments in expressive power are *exact*, as shown by *factorization theorems*: for example, every knowing strategy can be factored as the composition of a ground-type reference and an innocent strategy. This immediately reduces definability in the extended language with ground-type references to definability of innocent strategies by purely functional programs, as already shown in [8, 20]. Thus this method is technically powerful as well as exhibiting the surprisingly modular internal structure of the games universe.

(The relaxation of constraints on strategies is by no means the only meaningful variation of the game semantics setting which can be considered. In particular, the notion of game itself can be modified. A number of such modifications have already proved fruitful: allowing answers to enable questions, as in [12, 13], which is needed in order to model sum types; varying the way in which games can start, as in [3, 7], which allows call-by-value to be captured; and introducing a distinction between "active" and "passive" moves, used in giving a fully abstract semantics for pure Idealized Algol [5].)

The games we use to model general references are based on those which have been used in previous work [4, 7, 8, 11–13, 20], with one significant difference: we abandon the technical condition called *visibility*. Just as relaxing the condition of innocence corresponds to the model of ground-type references, so the additional freedom provided by abandoning visibility precisely captures the semantics of general references, as shown by our full abstraction result. The proof again uses factorisation arguments along the line of [4, 5, 11]. Each strategy in our model (which in general satisfies neither innocence nor visibility) can be factored into a higher-order cell and a strategy satisfying visibility; but in [4] it is shown that a strategy of the latter kind can be factored into the combination of a first-order cell and an innocent strategy. The definability result then follows immediately from the definability of innocent strategies [3, 7, 8, 20], leading to full abstraction. Thus the (lack of) visibility corresponds precisely to the semantic difference between higher-order cells and ground cells, offering a basic semantic analysis of general references.

2 A language with references

The language $\mathcal{L}$ we shall consider is the call-by-value simply typed λ-calculus over the ground type of natural numbers, extended with reference types. This is the simplest setting in which the key ideas behind our results can be conveyed. However, it is certainly possible to extend the model with richer functional features as in [14], and with control operators as in [11], yielding a model of a language with most of the features of Core ML [18]. We believe that the techniques used in the present paper, combined with those in [11,14], could be used to extend our results to this setting.

The types of our language are as follows.

$$A ::= \mathtt{unit} | \mathtt{nat} | A \times B | A \to B | \mathtt{var}[A].$$

The terms are those of the simply-typed λ-calculus (with products) extended with the constants for arithmetic and creation, assignment and dereferencing of program variables; see Figure 1, which also contains some typing rules. The `mkvar` construct allows for the creation of so-called "bad variables"—see Section 2.4. Note that we have not included recursion combinators; as we will see, these can be *defined*.

2.1 Operational semantics

The operational semantics of the language is given in terms of a countable collection of ***locations*** ranged over by $l, l_1, l_2, \ldots$. We give an inductive definition of a relation

$$(L, s)M \Downarrow (L', s')V$$

where M is an expression, V a value of the same type (see below), and (L, s) and (L', s') are ***stores*** consisting of a set L of location-type pairs and a partial function s from locations in L to values of the appropriate type. We write $s(l \mapsto V)$ for the store obtained by altering s so that location l is mapped to value V. The values of the language are: `skip`, numerals, λ-abstractions, pairs of values, `mkvar` $V\ V'$ where both V and V' are values, and locations.

A few clauses of the inductive definition are given in Figure 2; the remaining clauses are the standard call-by-value rules for function types, product types and arithmetic. We adopt the convention that, for example,

$$\frac{M \Downarrow V \quad M' \Downarrow V'}{M'' \Downarrow V''}$$

is an abbreviation for

$$\frac{(L, s)M \Downarrow (L', s')V \quad (L', s')M' \Downarrow (L'', s'')V'}{(L, s)M'' \Downarrow (L'', s'')V''}$$

to aid readability. For a closed term M containing no free locations, we write $M\Downarrow$ to indicate that $(\emptyset, \emptyset)M \Downarrow (L, s)V$ for some L, s and V. The notion of observational equivalence between terms is the usual one: $M \simeq N$ iff for all contexts $C[-]$ such that $C[M]$ and $C[N]$ are closed and contain no free locations, $C[M]\Downarrow$ iff $C[N]\Downarrow$.

2.2 Simple examples

The constructs of a basic imperative language can be considered as syntactic sugar for those of $\mathcal{L}$. The type `unit`, whose elements may have side-effects on the store but return no non-trivial values, is used as the type of commands. Sequential composition $C_1; C_2$ is syntactic sugar for

$$(\lambda d : \mathtt{unit}.C_2)C_1$$

where d is a variable not appearing in C_2. Note that this only works because evaluation in $\mathcal{L}$ is call-by-value. This construct applies equally when C_2 is not of type `unit`, yielding side-effecting expressions at all types. A variable declaration

$$\mathtt{new}\ x : A\ \mathtt{in}\ M$$

is sugar for

$$(\lambda x : \mathtt{var}[A].M)\mathtt{new}_A.$$

Iterative constructs such as `while` loops are desugared using recursion. We will define a Y combinator later, but for now let us show how a simple loop

$$\mathtt{while}\ M \neq 0\ \mathtt{do}\ C\ \mathtt{end}$$

is desugared. Intuitively, this ought to be a recursively defined element of type `unit`, but because $\mathcal{L}$ is call-by-value, it is necessary to work with "thunks" of type $\mathtt{unit} \to \mathtt{unit}$ instead. We therefore code such a loop as

$$\begin{array}{l} Y(\lambda f : (\mathtt{unit} \to \mathtt{unit}). \\ \quad \mathtt{ifzero}\ M\ \mathtt{then}\ (\lambda d : \mathtt{unit}.\mathtt{skip})\ \mathtt{else} \\ \quad\quad (\lambda d : \mathtt{unit}.C; f(\mathtt{skip}))) \end{array}$$

and apply the resulting value of type $\mathtt{unit} \to \mathtt{unit}$ to the dummy argument `skip` to execute it.

So far, our examples have been very simple and have not exploited the great power of general references. The definition of the Y combinator, on the other hand, does make use

$$
\begin{array}{rcl}
M & ::= & x \mid \texttt{skip} \mid n \mid \texttt{succ}\ M \mid \texttt{pred}\ M \mid \texttt{ifzero}\ M\ M\ M \mid \lambda x : A.M \mid MM \\
& \mid & \langle M, M\rangle \mid \texttt{fst}\ M \mid \texttt{snd}\ M \mid \texttt{new}_A \mid M := M \mid !\,M \mid \texttt{mkvar}\ M\ M.
\end{array}
$$

$$
\frac{}{\Gamma \vdash \texttt{new}_A : \texttt{var}[A]} \qquad \frac{\Gamma \vdash M : \texttt{var}[A]}{\Gamma \vdash !\,M : A}
$$

$$
\frac{\Gamma \vdash M : \texttt{var}[A] \quad \Gamma \vdash N : A}{\Gamma \vdash M := N : \texttt{unit}} \qquad \frac{\Gamma \vdash M : A \to \texttt{unit} \quad \Gamma \vdash N : \texttt{unit} \to A}{\Gamma \vdash \texttt{mkvar}\ M\ N : \texttt{var}[A]}
$$

Figure 1. Syntax and typing rules of $\mathcal{L}$

$$
\frac{M_1 \Downarrow V_1 \quad M_2 \Downarrow V_2}{\texttt{mkvar}\ M_1\ M_2 \Downarrow \texttt{mkvar}\ V_1\ V_2} \qquad \frac{}{(L, s)\texttt{new}_A \Downarrow (L \cup \{l : A\}, s)l}\ l \notin L
$$

$$
\frac{(L, s)M \Downarrow (L', s')l \quad (L', s')N \Downarrow (L'', s'')V}{(L, s)M := N \Downarrow (L'', s(l \mapsto V))\texttt{skip}} \qquad \frac{M \Downarrow \texttt{mkvar}\ V_1\ V_2 \quad N \Downarrow V \quad V_1(V) \Downarrow \texttt{skip}}{M := N \Downarrow \texttt{skip}}
$$

$$
\frac{(L, s)M \Downarrow (L', s')l \quad s'(l) = V}{(L, s)\,!\,M \Downarrow (L', s)V} \qquad \frac{M \Downarrow \texttt{mkvar}\ V_1\ V_2 \quad V_2(\texttt{skip}) \Downarrow V}{!\,M \Downarrow V}
$$

Figure 2. Operational semantics of $\mathcal{L}$

of this expressiveness. It is well known that general references can be used to build cyclic structures in the store. One consequence of this is that recursively defined functions can be encoded by building the corresponding cyclic graph. We can therefore define a Y combinator as follows.

$$
\begin{array}{l}
\lambda f : (A \to B) \to (A \to B). \\
\ \texttt{new}\ y : A \to B\ \texttt{in} \\
\ \ y := \lambda a : A.f(!\,y)(a); \\
\ \ \texttt{return}\ !\,y.
\end{array}
$$

To implement a recursively defined function, this program generates a new reference y, builds in it a cyclic graph, and returns the value so created. Note that the type of this operator does not contain any reference types. This means that it is impossible for the environment to alter the value stored in y.

2.3 Objects

Leaving aside issues of sub-typing and inheritance, the *behavioural* aspects of objects can be represented in a direct and general fashion in our language. The general scheme is that an object is represented by a term of the form

$$
\texttt{new}\ l_1 := v_1, \ldots l_k := v_k\ \texttt{in}\ \langle m_1, \ldots, m_p\rangle
$$

where $l_1, \ldots, l_k$ are the local variables of the object, and $m_1, \ldots, m_k$ are functions representing its methods; and $\texttt{new}\ x\ \texttt{in}\ C$ is syntactic sugar for $(\lambda x.C)\texttt{new}$. The type of such an object will simply be the product of the types of its methods. (Later we will use a similar idea to model references themselves in our semantic universe.) For example, a stack object might be represented thus:

$$
\begin{array}{l}
\texttt{new}\ p : \texttt{nat} := 0, a : \texttt{array[]}\ \texttt{of}\ \texttt{nat}\ \texttt{in} \\
[\texttt{push} = \lambda n : \texttt{nat}.p := !\,p + 1; a[!\,p] := n, \\
\ \texttt{pop} = \lambda().\texttt{if}\ !\,p \neq 0\ \texttt{then}\ p := !\,p - 1, \\
\ \ldots \text{etc.} \\
]
\end{array}
$$

Here `array[] of nat` is syntactic sugar for a type of array objects, which can readily be programmed in $\mathcal{L}$. Note that the locality of the local state of the object is enforced simply by the fact that the variables are bound by `new` and do not appear in its type. Moreover, any (acyclic) object hierarchy, in which objects can have objects in their local state, as above, and methods can take objects as arguments and return objects as results, can be accommodated within the framework of simple types using this representation.

Of course, we don't claim that this offers a full analysis of object-oriented programming; but the way in which the behavioural aspects can be represented so directly does suggest the expressive power of higher-type references.

2.4 Equality of references and bad variables

One important aspect of reference types in languages such as Standard ML is *not* reflected in our language; namely the operation of comparing references for equality. This operation is intimately related with the issue of *bad variables*, an issue first identified by Reynolds [22]. Roughly speaking, a bad variable is a value of reference type which does not denote a location in the store. It clearly makes no sense to test whether two such things denote the same location! On the other hand, if all variables are "good", then equality of (non-trivial) references is *definable*. (The idea for ground types is simply to write different values into the two variables, and then test if they contain the same value. This can be extended to higher types using constant functions, as pointed out to the first author by John Reynolds; although, intriguingly, it does not extend to the type of references which cannot hold interesting values, $\mathtt{var}[\mathtt{unit}]$, which is essentially the type of *names* of the π-calculus.)

We shall follow Reynolds in viewing the type $\mathtt{var}[A]$ as a product of a "read method" and a "write method". Once this view is taken, then behaviours of type $\mathtt{var}[A]$ need look nothing like *bona fide* references (for example, reads need not be causally related to writes). Such bad variables are created in our language using the `mkvar` construct, which takes a value of type $A \to \mathtt{unit}$ (the write method), and a value of type $\mathtt{unit} \to A$ (the read method), returning a "variable object" of type $\mathtt{var}[A]$. Thus `mkvar` makes explicit the identification of the type $\mathtt{var}[A]$ with the product of its method types.

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, Peter O'Hearn has pointed out that in the *absence* of bad variables, the constructor $\mathtt{var}[\cdot]$ is not functorial! From the point of view of semantics, therefore, bad variables appear to make the path smoother, and this is certainly the case for the present paper: our definability result makes essential use of `mkvar`. Finding a fully abstract model for a language with equality of references must be left as a challenge for future work.

3 A games model of references

We now define the categories of games which provide our model of $\mathcal{L}$ and explain how $\mathcal{L}$ is interpreted. Our interpretation is structured using a *computational monad* [19] which serves to distinguish values, that is, programs which terminate immediately without accessing the store, from computations, which may read from and write to the store before terminating, and may also fail to terminate at all.

The games and strategies we use are direct descendants of those used by Hyland, Ong and Nickau [8, 20] to provide fully abstract models of PCF. The main differences are that the key technical conditions of *visibility* and *innocence* (see below) are both relaxed. Another difference is that we model call-by-value, rather than call-by-name, computation. For this we can use either the construction in [7] or that of [3], with the same effect; here we choose the latter.

3.1 Arenas

A game has two participants: Player (P) and Opponent (O). A *play* of the game consists of a sequence of moves, alternately by O and P. In addition, each move is explicitly *justified* by an earlier move of the play, unless it is a special kind of move, called *initial*, which needs no justification. In the games we consider, O always moves first.

Before embarking on a formal definition, let us fix notation for sequences and operations on them. We use $s, t, \ldots$ to range over sequences and $a, b, \ldots$ over elements of sequences. If s and t are sequences, then st or $s \cdot t$ is their concatenation; ε is the empty sequence. A move a will often be identified with the singleton sequence consisting just of a.

An ***arena*** is specified by a triple $A = \langle M_A, \lambda_A, \vdash_A \rangle$ where

- M_A is a set of ***moves***.
- $\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$ is a ***labelling*** function which indicates whether a move is by Opponent (O) or Player (P), and whether it is a question (Q) or an answer (A). We write

$$\begin{array}{rcl} \{\mathsf{O},\mathsf{P}\} \times \{\mathsf{Q},\mathsf{A}\} & = & \{\mathsf{OQ},\mathsf{OA},\mathsf{PQ},\mathsf{PA}\} \\ \lambda_A & = & \langle \lambda_A^{\mathsf{OP}}, \lambda_A^{\mathsf{QA}} \rangle. \end{array}$$

  The function $\overline{\lambda}_A$ is λ_A with the O/P part reversed, so that
$$\overline{\lambda}_A(a) = \mathsf{OQ} \iff \lambda_A(a) = \mathsf{PQ}$$
  and so on. If $\lambda^{\mathsf{OP}}(a) = \mathsf{O}$, we call a an O-move; otherwise, a is a P-move.
- $\vdash_A$ is a relation between $M_A + \{\star\}$ and M_A, called ***enabling***, which satisfies
  - $\star \vdash_A a \Rightarrow \lambda_A(a) = \mathsf{OQ} \wedge$
    $[b \vdash_A a \iff b = \star]$.
  - $a \vdash_A b \wedge \lambda_A^{\mathsf{QA}}(b) = \mathsf{A} \Rightarrow \lambda_A^{\mathsf{QA}}(a) = \mathsf{Q}$.
  - $a \vdash_A b \wedge a \neq \star \Rightarrow \lambda_A^{\mathsf{OP}}(a) \neq \lambda_A^{\mathsf{OP}}(b)$.

The enabling relation tells us either that a move a is ***initial*** and needs no justification ($\star \vdash_A a$), or that it can be justified by another move b, if b has been played ($b \vdash_A a$).

A ***justified sequence*** s of moves in an arena A is a sequence of moves together with ***justification pointers***: for each move a in s which is not initial, there is a pointer to an earlier move b of s such that $b \vdash_A a$. We say the move b

justifies a, and extend this terminology to say that a move b ***hereditarily justifies*** a if the chain of pointers back from a passes through b.

A justified sequence is a ***legal position*** iff:

- s is ***alternating***: if $s = s_1 abs_2$ then $\lambda^{\mathsf{OP}}(a) \neq \lambda^{\mathsf{OP}}(b)$.
- The ***bracketing condition*** holds. We say that a question q in s is ***answered*** by a later answer a in s if q justifies a. The bracketing condition is satisfied by s if for each prefix $t \cdot \overset{\frown}{q \cdot u \cdot a}$ of s, all questions asked in u are answered within u; in other words, when an answer is given, it is always to the most recent question which has not been answered—the ***pending question***.

The set of all legal positions of an arena A is written L_A.

3.2 Strategies

A strategy for an arena A is a rule telling Player what move to make in a given position. Formally, this can be represented as a set σ of legal positions in which P has just moved, i.e. a set of even-length positions, such that

- $\varepsilon \in \sigma$
- $sab \in \sigma \Rightarrow s \in \sigma$
- $sab, sac \in \sigma \Rightarrow b = c$.

For any arena A, the smallest possible strategy is $\{\varepsilon\}$, which never makes any response. It is called the ***empty strategy*** and denoted $\perp$.

Given a non-empty legal position sa in an arena A, the ***current thread*** $\mathsf{thread}(sa)$ is the subsequence of sa containing all moves hereditarily justified by the same initial move as a. A strategy σ for A is ***thread-independent*** iff

- if $sab \in \sigma$ then b is justified by a move in $\mathsf{thread}(sa)$; and
- if $sab, t \in \sigma$, and $ta \in L_A$ is such that $\mathsf{thread}(sa) = \mathsf{thread}(ta)$, then $tab \in \sigma$, with the justification pointer on b such that $\mathsf{thread}(tab) = \mathsf{thread}(sab)$.

That is to say, a thread-independent strategy chooses its move at a position sa based just on the moves in the current thread $\mathsf{thread}(sa)$. From now on we will only be interested in thread-independent strategies. We write $\sigma : A$ to indicate that σ is a thread-independent strategy for A.

3.3 Constructions on arenas

Given arenas A and B, the arenas $A \times B$ and $A \Rightarrow B$ are defined as follows.

$$\begin{array}{rcl} M_{A\times B} & = & M_A + M_B \text{ (disjoint union)} \\ \lambda_{A\times B} & = & [\lambda_A, \lambda_B] \\ \star \vdash_{A\times B} a & \iff & \star \vdash_A a \vee \star \vdash_B a \\ a \vdash_{A\times B} b & \iff & a \vdash_A b \vee a \vdash_B b \\ M_{A\Rightarrow B} & = & M_A + M_B \\ \lambda_{A\Rightarrow B} & = & [\overline{\lambda}_A, \lambda_B] \\ \star \vdash_{A\Rightarrow B} a & \iff & \star \vdash_B a \\ a \vdash_{A\Rightarrow B} b & \iff & a \vdash_A b \vee a \vdash_B b \vee \\ & & [\star \vdash_B a \wedge \star \vdash_A b] \end{array}$$

The idea here is that the games A and B are played in parallel. In $A \times B$, labelling and enabling are inherited directly from A and B. In $A \Rightarrow B$, the O/P roles in A are reversed, and initial moves of A are now justified by initial moves of B. The unit for $\times$ is the empty arena $\mathbf{1} = \langle \emptyset, \emptyset, \emptyset \rangle$.

It is worth noting that in contrast to other games models, it is *not* the case that if s is a legal position of $A \times B$, then the subsequence $s \restriction A$ of moves from A is a legal position of A. Indeed, $s \restriction A$ may not even be alternating. This liberal approach is necessary for our interpretation of higher-order store.

3.4 The category $\mathcal{C}$

We define a category $\mathcal{C}$ as follows.

Objects	:	Arenas.
Morphisms $A \to B$	:	Thread-independent strategies for $A \Rightarrow B$.

Identity The identity map $\mathsf{id} : A \to A$ on an arena A is given as usual by the 'copycat' strategy

$$\{s \in P_{A_1 \Rightarrow A_2} \mid \forall t \sqsubseteq^{\text{even}} s.\, t \restriction A_1 = t \restriction A_2\}.$$

We use subscripts on the 'A's to distinguish the two occurrences, and write $t \sqsubseteq^{\text{even}} s$ to mean that t is an even-length prefix of s. This strategy responds to any move simply by copying it to the other component.

Composition Given maps $\sigma : A \to B$ and $\tau : B \to C$, i.e. strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, we define

$$\sigma \parallel \tau = \left\{ s \;\middle|\; \begin{array}{l} s \restriction A, B \in \sigma, s \restriction B, C \in \tau, \\ s \restriction A, C \in L_{A\Rightarrow C} \end{array} \right\}$$

where s ranges over sequences of moves from $M_A + M_B + M_C$. We then set

$$\sigma\,;\tau = \{s \restriction A, C \mid s \in \sigma \parallel \tau\}.$$

This can be shown to be well-defined and associative.

3.5 $\mathcal{C}$ as a cartesian closed category

The operation $\times$ is the categorical product in $\mathcal{C}$. The projections are given by the obvious copy-cat strategies $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. Given thread-independent strategies $\sigma : C \to A$ and $\tau : C \to B$, their pairing $\langle \sigma, \tau \rangle : C \to A \times B$ is defined as follows.

If $s \in L_{C \Rightarrow A \times B}$, write s_1 for the subsequence of those moves hereditarily justified by an initial move of A, and s_2 for those moves hereditarily justified by an initial move of B. Then $\langle \sigma, \tau \rangle$ is defined by:

$$\{s \in L_{C \Rightarrow A \times B} \mid s_1 \in \sigma, s_2 \in \tau\}.$$

It is straightforward to check that this does indeed give rise to a product in $\mathcal{C}$.

For any $\sigma : A \times B \to C$, there is a strategy $\Lambda(\sigma) : A \to (B \Rightarrow C)$ defined by relabelling the moves of σ. It can be checked that this bijection of hom-sets makes $\mathcal{C}$ into a cartesian closed category.

3.6 Weak coproducts

In addition to its cartesian closed structure, $\mathcal{C}$ also has a weak coproduct which, despite not being a genuine coproduct, enjoys a universal property analogous to that of the separated sum of cpos. To state this, some auxiliary definitions are necessary.

Say that an arena A is ***pointed*** if it has exactly one initial move. Given pointed arenas A and B, a map $\sigma : A \to B$ of $\mathcal{C}$ is ***strict*** iff it responds to the initial move of B with the initial move of A, which it never plays again. Pointed arenas and strict maps form a subcategory $\mathcal{C}_\perp$. In fact $\mathcal{C}_\perp$ is an *exponential ideal*: if B is pointed, then so is $A \Rightarrow B$ for any A, and if τ is a strict map, then so is $\sigma \Rightarrow \tau$ for any σ.

Let $\{A_i \mid i \in I\}$ be a family of arenas. The pointed arena $\Sigma_{i \in I} A_i$ has as its set of moves

$$\{q\} + \{i \mid i \in I\} + \Sigma_i M_{A_i}$$

where $\Sigma_i M_{A_i}$ denotes disjoint union of sets. The move q is the unique initial question, asking which of the A_i to play in. There is one answer to q for each $i \in I$; this answer in turn enables the initial moves of A_i. For all other moves, labelling and enabling is inherited from the A_i.

For each i there is an obvious (non-strict) strategy $\mathsf{in}_i : A_i \to \Sigma_{i \in I} A_i$ which responds to the initial question with the move i and thereafter plays copycat. The universal property of the sum can now be stated as follows.

For any family $\{\sigma_i : A_i \to B \mid i \in I\}$ of maps in $\mathcal{C}$, with B pointed, there is a unique map $[\sigma_i \mid i \in I] : \Sigma_i A_i \to B$ in $\mathcal{C}_\perp$ such that for each i, $\mathsf{in}_i ; [\sigma_i \mid i \in I] = \sigma_i$. The strategy $[\sigma_i \mid i \in I]$ behaves as follows. It responds to the unique initial question of B by asking q in $\Sigma_i A_i$. After O responds with some i, it continues to play as σ_i would play, ignoring the extra two moves $q \cdot i$.

The separated sum of a singleton family $\{A\}$ corresponds to lifting; we shall therefore write this as $A_\perp$. The unique answer to the initial question q will be written as $*$.

Remark The condition of thread-independence depends crucially on the initial moves of an arena; since the weak coproduct and lifting constructions add new initial moves, they have a great effect on the available thread-independent strategies. For example, a thread-independent strategy for $A \times B$ must consist of a pair of thread-independent strategies, one for A and one for B. Thread-independence implies that no information flows between these two strategies. On the other hand, a thread-independent strategy for $(A \times B)_\perp$ may be of a very different kind: moves made in A may affect later moves in B and vice versa. This can be seen as reflecting the difference between values of type $A \times B$, which are pairs of values for A and B, and arbitrary terms, which may create and use store, allowing the A and B parts to communicate. In the following section, these distinctions will be packaged in the form of a strong monad.

3.7 The category $\mathrm{Fam}(\mathcal{C})$

The category $\mathcal{C}$ defined above most naturally models call-by-name programming languages, but $\mathcal{L}$ is call-by-value. We therefore require certain modifications. We choose to use the construction given in [3], so that our model of $\mathcal{L}$ lives not in $\mathcal{C}$ but in its free completion under coproducts, which we call $\mathrm{Fam}(\mathcal{C})$. An alternative, more direct, presentation of the call-by-value model could be given along the lines of [7].

The objects of $\mathrm{Fam}(\mathcal{C})$ are families $\{A_i \mid i \in I\}$ of arenas. A map from $\{A_i \mid i \in I\}$ to $\{B_j \mid j \in J\}$ consists of a reindexing function $f : I \to J$ together with a family of maps $\{\sigma_i : A_i \to B_{f(i)} \mid i \in I\}$ of $\mathcal{C}$; thus each σ_i is a thread-independent strategy. As shown in [3], the cartesian closed structure of $\mathcal{C}$, together with the fact that $\mathcal{C}$ has *all* small products, makes $\mathrm{Fam}(\mathcal{C})$ cartesian closed, and the weak coproduct structure gives rise to a strong monad T on $\mathrm{Fam}(\mathcal{C})$. Given families $A = \{A_i \mid i \in I\}$ and $B = \{B_j \mid j \in J\}$, we set

$$\begin{array}{rcll} A \times B & = & \{A_i \times B_j \mid (i,j) \in I \times J\} & \\ A \Rightarrow B & = & \{\Pi_{i \in I}(A_i \Rightarrow B_{f(i)}) \mid f : I \to J\} & \\ TA & = & \{\Sigma_{i \in I} A_i\}, & \text{a singleton family.} \end{array}$$

The family $\{(\mathsf{in}_i : A_i \to \Sigma_i A_i)_i \mid i \in I\}$ gives the unit $\eta : A \to TA$ of the monad, and the copairing operation of the weak coproduct is used to define the "lifting" operation of the monad, taking a map $f : A \times B \to TC$ to $f^* : A \times TB \to TC$. There are two distinct ways of using this operation to turn the unit $\eta : A \times B \to T(A \times B)$ into so-called "double strength" morphisms

$$\mathsf{dst}, \mathsf{dst}' : TA \times TB \to T(A \times B).$$

Intuitively, dst produces a computation of type $A \times B$ which works by evaluating its arguments, computations of type A and B, in left-to-right order, while dst' evaluates right-to-left. The inequality of dst and dst' reflects the fact that the order in which side-effecting programs are evaluated is critical. In the categorical jargon, this inequality means that the monad T is not *commutative* [10].

$\mathrm{Fam}(\mathcal{C})$ also has coproducts given by disjoint union of families. The object $\mathbf{N} = \{\mathbf{1}_n \mid n \in \omega\}$ is a natural numbers object.

3.8 Constraining strategies

We will make use of some additional constraints which can be applied to strategies, both of which depend on the notion of view of a justified sequence. Given an odd-length justified sequence s, its ***view*** $\ulcorner s \urcorner$ is defined as follows.

$$\begin{array}{rcl} \ulcorner s \cdot a \urcorner & = & a, \text{if } a \text{ is initial} \\ \ulcorner s \cdot \overset{\frown}{a \cdot t \cdot b} \urcorner & = & \ulcorner s \urcorner \cdot \overset{\frown}{a \cdot b}. \end{array}$$

A strategy $\sigma : A$ satisfies the ***visibility condition*** iff for all $sab \in \sigma$, the move justifying b lies in $\ulcorner sa \urcorner$. Given $\sigma : A$ satisfying the visibility condition, we say σ is ***innocent*** iff

$$\forall sab, t \in \sigma . ta \in L_A \wedge \ulcorner sa \urcorner = \ulcorner ta \urcorner \Rightarrow tab \in \sigma,$$

and the justification pointer from b in tab points to the same move of $\ulcorner ta \urcorner$ as does the pointer from b in sab. That is to say, an innocent strategy bases its response at a position sa just on the view $\ulcorner sa \urcorner$. (Note that an innocent strategy is always thread-independent.) The identity strategy is innocent, and both innocence and visibility are preserved by composition. The subcategory of innocent strategies provides a fully abstract model of pure functional languages like PCF [8, 20], while strategies subject to the visibility condition yield full abstraction for languages with first-order store [3, 4].

3.9 Interpretation of $\mathcal{L}$

We now briefly review the standard interpretation of a call-by-value language in a CCC with strong monad [19], which forms the backbone of our interpretation of $\mathcal{L}$ in $\mathrm{Fam}(\mathcal{C})$. The interpretation of the type $\mathtt{var}[-]$ and constants relating to store is deferred to the next section.

Each type A of $\mathcal{L}$ will be interpreted as an object $[\![A]\!]$. The interpretations of `unit` and `nat` are $\mathbf{1}$ and $\mathbf{N}$ respectively; the denotations of product and function types are defined inductively:

$$\begin{array}{rcl} [\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] & = & [\![A]\!] \Rightarrow T[\![B]\!]. \end{array}$$

A term $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ is interpreted as a map $[\![A_1]\!] \times \cdots \times [\![A_n]\!] \to T[\![A]\!]$. A few clauses of the interpretation are given in Figure 3, in which succ denotes the successor map of the natural numbers object $\mathbf{N}$, and Γ ranges over contexts of the form $x_1 : A_1, \ldots, x_n : A_n$.

3.10 Modelling store in $\mathrm{Fam}(\mathcal{C})$

To make $\mathrm{Fam}(\mathcal{C})$ into a model of $\mathcal{L}$, we must give an object interpreting the type $\mathtt{var}[A]$, and morphisms to interpret the terms relating to store. Following Reynolds [23] (cf. [16]), we take an "object oriented" view of variables, modelling the type $\mathtt{var}[A]$ as the product of its "write method", of type $A \Rightarrow T\mathbf{1}$, and its "read method", of type TA. We therefore set

$$[\![\mathtt{var}[A]]\!] = ([\![A]\!] \Rightarrow T\mathbf{1}) \times T[\![A]\!].$$

The first and second projections give rise to maps

$$[\![\mathtt{var}[A]]\!] \times [\![A]\!] \xrightarrow{\mathsf{assign}} T[\![\mathtt{unit}]\!]$$
$$[\![\mathtt{var}[A]]\!] \xrightarrow{\mathsf{deref}} T[\![A]\!]$$

and hence to interpretations of assignment and dereferencing, while `mkvar` can be interpreted using pairing; see Figure 4.

It remains to give an interpretation of $\mathtt{new}_A$. For each $A = \{A_i \mid i \in I\}$, we require a morphism $\mathbf{1} \to T\mathtt{var}[A]$, that is, a strategy cell for the game

$$(\Pi_{i \in I}(A_i \Rightarrow \mathbf{1}_\perp) \times \Sigma_{i \in I} A_i)_\perp.$$

We shall write the initial move of the $\Sigma_i A_i$ component as read, since it corresponds to a request to read from the variable; similarly, the initial question of the ith component of $\Pi_i(A_i \Rightarrow T\mathbf{1})$ will be called $\mathsf{write}(i)$, and its unique answer is ok. The strategy cell has the following behaviour.

- It responds to the initial question q with its unique answer $*$.
- If the next move is read, it makes no response (so cell corresponds to an *uninitialized* variable).
- cell responds to each $\mathsf{write}(i)$ with ok.
- If the move read is played and the most recently played write move was $\mathsf{write}(i)$, cell responds with i.
- Suppose O plays a move a initial in some A_i in the read component. This move must be justified by some answer i to the move read. cell copies the move a to the write component, justified by the most recent $\mathsf{write}(i)$ which occurred before the read. (This violates the visibility condition).
- Thereafter, cell copies all moves hereditarily justified by either of these copies of a back and forth between the read and write components.

Each time O plays a read, it is establishing a new "read thread"; similarly each $\mathsf{write}(i)$ establishes a "write thread". When O plays a read, cell creates a "link" between the current read thread and the most recently established write thread. Concretely, this link is realized by copying moves between the read thread and the write thread, in a fashion by now familiar in game semantics. The usual, "logical" links interpreted by copy-cat strategies which are used to build

$$
\begin{aligned}
[\![\Gamma \vdash x_i : A_i]\!] &= \pi_i \,;\eta : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to T[\![A_i]\!] \\
[\![\Gamma \vdash \lambda x.M : A \to B]\!] &= \Lambda([\![\Gamma, x : A \vdash M : B]\!])\,;\eta : [\![\Gamma]\!] \to T([\![A]\!] \to T[\![B]\!]) \\
[\![\Gamma \vdash MN : B]\!] &= \langle [\![\Gamma \vdash M : A \to B]\!], [\![\Gamma \vdash N : A]\!] \rangle \,; \mathsf{dst}\,; \mathsf{ev}^* : [\![\Gamma]\!] \to T[\![B]\!] \\
[\![\Gamma \vdash \mathtt{succ}\ M : \mathtt{nat}]\!] &= [\![\Gamma \vdash M]\!]\,; T\mathsf{succ} : [\![\Gamma]\!] \to T[\![\mathtt{nat}]\!]
\end{aligned}
$$

Figure 3. Interpretation of $\mathcal{L}$

$$
\begin{aligned}
[\![\Gamma \vdash M := N : \mathtt{unit}]\!] &= \langle [\![\Gamma \vdash M : \mathtt{var}[A]]\!], [\![\Gamma \vdash N : A]\!] \rangle \,; \mathsf{dst}\,; \mathsf{assign}^* : [\![\Gamma]\!] \to T[\![\mathtt{unit}]\!] \\
[\![\Gamma \vdash\, !\,M : A]\!] &= [\![\Gamma \vdash M : \mathtt{var}[A]]\!]\,; \mathsf{deref}^* : [\![\Gamma]\!] \to T[\![A]\!] \\
[\![\Gamma \vdash \mathtt{mkvar}\ M\ N : \mathtt{var}[A]]\!] &= \langle [\![\Gamma \vdash M : A \to \mathtt{unit}]\!], [\![\Gamma \vdash N : \mathtt{unit} \to N]\!] \rangle \,; \mathsf{dst}\,; \cong : [\![\Gamma]\!] \to T[\![\mathtt{var}[A]]\!]
\end{aligned}
$$

Figure 4. Semantics of store

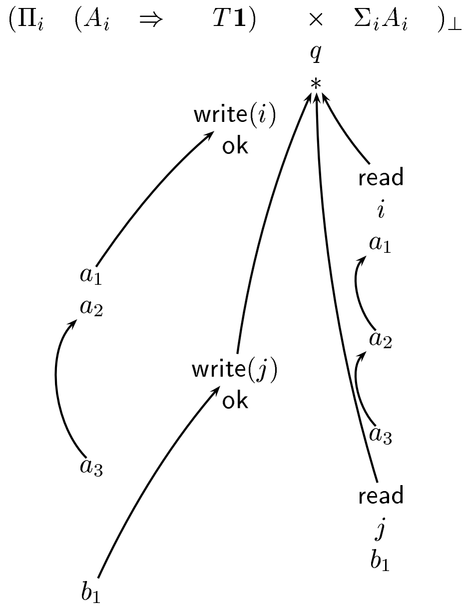


Figure 5. A typical play of cell

the cartesian closed structure of our model are statically determined by the types. By contrast, the links established by cell are highly dynamic, and violate behavioural constraints such as visibility.

Figure 5 depicts a typical play of cell. An omitted justification pointer on a non-initial move indicates that the move is justified by the immediately preceding one. Notice how the justification pointers serve to indicate which particular thread of writing is copied to each thread of reading.

This completes our description of the game semantics of $\mathcal{L}$. We now state a simple lemma which shows that the order of (disjoint) allocations and assignments of variables does not matter, and that assignment to a variable x destroys any value previously stored there.

Lemma 1 The equations given in Figure 6 hold in the games model of $\mathcal{L}$ whenever the terms concerned are well-typed.

4 Soundness and Adequacy

The first soundness result that we require is that the denotational semantics of $\mathcal{L}$ respects its operational semantics in an appropriate way. Before stating the result, let us introduce some notation. Given a store (L, s) and a term

$$l_1 : \mathtt{var}[A_1], \ldots, l_n : \mathtt{var}[A_n] \vdash M : A$$

where the l_i and their types correspond to the locations and types in L, we write $\mathtt{new}\ L, s\ \mathtt{in}\ M$ for the expression $\mathtt{new}\ l_1 : A_1, \ldots, l_n : A_n\ \mathtt{in}\ l_1 := s(l_1); \ldots; l_n := s(l_n); M$. (The order of these allocations and assignments does not matter because of Lemma 1.) The soundness of our model can then be stated as follows.

Proposition 2 (Soundness) If for some term M we have $(L, s)M \Downarrow (L', s')V$, then $[\![\mathtt{new}\ L, s\ \mathtt{in}\ (\lambda x.N)M]\!] = [\![\mathtt{new}\ L', s'\ \mathtt{in}\ (\lambda x.N)V]\!]$ for any suitably typed term N.

This is proved by a simple induction on the derivation of $(L, s)M \Downarrow (L', s')V$, using standard facts about strong monads on CCCs together with the equations of Figure 6 and the additional equation

$$
\begin{aligned}
& [\![\Gamma \vdash \mathtt{new}\ x : A\ \mathtt{in}\ x := V; (\lambda y.M)(!\,x)]\!] \\
= \ & [\![\Gamma \vdash \mathtt{new}\ x : A\ \mathtt{in}\ x := V; (\lambda y.M)(V)]\!].
\end{aligned}
$$

The validity of this final equation is proved by a detailed analysis of the strategies involved, showing that the copycat

$$
\begin{array}{rcl}
[\![\Gamma \vdash \mathtt{new}\, x : A, y : B \,\mathtt{in}\, M]\!] & = & [\![\Gamma \vdash \mathtt{new}\, y : B, x : A \,\mathtt{in}\, M]\!] \\
[\![\Gamma, x : \mathtt{var}[A] \vdash \mathtt{new}\, y : B \,\mathtt{in}\, x := V; M]\!] & = & [\![\Gamma, x : \mathtt{var}[A] \vdash x := V; \mathtt{new}\, y : B \,\mathtt{in}\, M]\!] \\
[\![\Gamma \vdash \mathtt{new}\, x : A, y : B \,\mathtt{in}\, x := V_1; y := V_2; M]\!] & = & [\![\Gamma \vdash \mathtt{new}\, x : A, y : B \,\mathtt{in}\, y := V_2; x := V_1; M]\!] \\
[\![\Gamma \vdash \mathtt{new}\, x : A \,\mathtt{in}\, x := V_1; x := V_2; M]\!] & = & [\![\Gamma \vdash \mathtt{new}\, x : A \,\mathtt{in}\, x := V_2; M]\!]
\end{array}
$$

Figure 6. Equations concerning assignments and allocations

behaviour of cell correctly feeds moves from the "writer" to the "reader".

We also need:

Proposition 3 (Adequacy) For any closed term M, if $[\![M]\!] \neq \bot$ then $M\Downarrow$.

The (long) proof works by analysing the interactions involved in a given non-empty play of the strategy $[\![M]\!]$: such a play arises from the interaction of several plays of strategies representing subterms of M. The length of this interaction forms the basis for an inductive proof. Putting these results together, we have:

Theorem 4 (Equational Soundness) If M and N are terms of the same type and $[\![M]\!] = [\![N]\!]$, then $M \simeq N$.

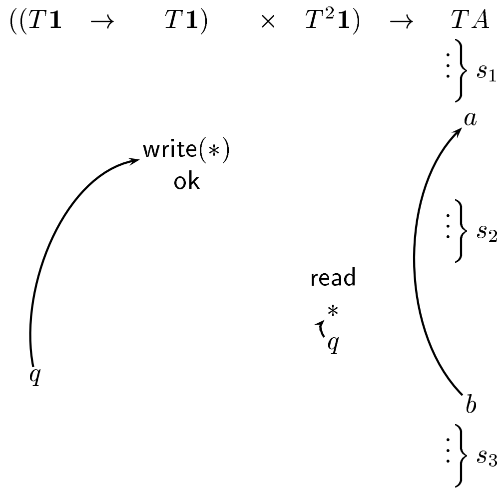
5 Full abstraction

Our proof of definability, and hence full abstraction, hinges on a factorization result, which we shall now prove. Throughout this section we will often use the syntax of $\mathcal{L}$ as an informal notation for the constructions on strategies defined by the semantics, and we will identify the object $[\![\mathtt{unit} \to \mathtt{unit}]\!]$ with the isomorphic $T\mathbf{1}$.

Proposition 5 For any arena A and any finite, thread-independent strategy $\sigma : \mathbf{1} \to TA$, there exists a natural number k and thread-independent strategy $\tau : \overbrace{\mathtt{var}[T\mathbf{1}] \times \cdots \times \mathtt{var}[T\mathbf{1}]}^{k} \to TA$ such that τ is finite and satisfies the visibility condition, and

$$(\mathtt{new}\, x_1, \ldots, x_k : \mathtt{unit} \to \mathtt{unit}\ \mathtt{in}\ \tau) = \sigma.$$

Proof We will just consider the case where σ consists of all even-length prefixes of a single sequence s, which may contain violations of the visibility condition. The argument generalizes to arbitrary finite σ straightforwardly. Suppose $s = s_1 a \overleftarrow{s_2} b s_3$ where b is a P-move and $s_1 a \overleftarrow{s_2} b$ violates visibility. Consider the following justified sequence in $\mathtt{var}[T\mathbf{1}] \Rightarrow TA$.



The play in TA is the same as s, while the play in $\mathtt{var}[T\mathbf{1}]$ follows the pattern of cell. Most importantly, the move b no longer violates the visibility condition, since a is in the view when b is played. This transformation is the essence of the factorization result; however, the above sequence need not satisfy the bracketing condition, and the transformation may cause moves in s_3 to violate the visibility condition where they did not before, so some care is needed.

We now describe the transformation precisely. Let k be the number of O-moves in s; we will produce a legal position in the arena $\underbrace{\mathtt{var}[T\mathbf{1}] \times \ldots \times \mathtt{var}[T\mathbf{1}]}_{k} \to TA$ which satisfies visibility and is identical to s in the TA component. First insert the segment write$(*)$ok after each O-move of s, the nth such segment being in the nth $\mathtt{var}[T\mathbf{1}]$ component. We now insert further sequences of moves before each P-move, starting from the first.

Case 1 Before a P-question, justified by the nth O-move, insert the segment read $\cdot * \cdot q \cdot q$, in the nth $\mathtt{var}[T\mathbf{1}]$ component, as above. The final q is justified by the write$(*)$ inserted immediately after the nth O-move, so the following P-move satisfies visibility; it also satisfies bracketing since it is a question.

Case 2 Before a P- answer, the play so far will take the form (writing q for questions and a for answers)

$$\ldots q \cdot t_n \cdot q_n \cdots a_n \cdots t_2 \cdot q_2 \cdots a_2 \cdot t_1 \cdot q_1 \cdots a_1$$

where q is the question to be answered by the following P-move and the t_i are the moves in the $\mathtt{var}[T\mathbf{1}]$ components we have so far inserted, which therefore take the form write$(*)$ok in some component, followed by read $\cdot * \cdot q \cdot q$ in some other component. We insert the following sequence of moves: P answers the final q of t_1, then O copies this answer to the previous q; this is repeated for $t_2, \ldots, t_n$. Thus when the following P-answer is played, neither bracketing nor visibility are violated.

Taking all even length prefixes of this sequence yields the required strategy τ. □

This factorization result reduces the question of definability for finite strategies to that for strategies satisfying the visibility condition. Another factorization result, taken from [4], is of further assistance.

Proposition 6 Let $\tau : \mathbf{1} \to TA$ be a finite, thread-independent strategy satisfying the visibility condition. Then there is a strategy $\upsilon : \mathtt{var}[\mathbf{N}] \to TA$ such that υ is innocent, has finite view-function, and satisfies

$$(\mathtt{new}\ x : \mathtt{nat}\ \mathtt{in}\ \tau) = \upsilon.$$

(The ***view-function*** of an innocent strategy is the function from views to P-moves together with justification pointers which determines the strategy.) The idea behind the proof of this result is that υ uses a coding of positions as natural numbers to record the history of play in its storage cell, so that it can simulate the history-sensitive strategy τ while still being innocent.

The question of definability of arbitrary finite strategies has now been reduced to that for innocent strategies with finite view-function; but this question has been answered in the affirmative by previous results on full abstraction for functional languages [7,8,12,20], so we have the following.

Theorem 7 Every finite morphism $\sigma : \mathbf{1} \to TA$ in the model of $\mathcal{L}$ is definable.

Proof The above factorization results mean that σ can be written in the form

$$\mathtt{new}\ x_1, \ldots, x_k : \mathtt{unit} \to \mathtt{unit}\ \mathtt{in}\ \mathtt{new}\ x : \mathtt{nat}\ \mathtt{in}\ \upsilon$$

for some innocent strategy υ with finite view function. The definability results for innocent strategies imply that $\upsilon = [\![M]\!]$ for some term M, essentially a pure PCF term. (Our identification of $\mathtt{var}[A]$ with $(A \to \mathtt{unit}) \times (\mathtt{unit} \to A)$, and the mkvar construct, are vital here.) Hence σ is definable by

$$(\lambda x_1, \ldots, x_k : \mathtt{var}[\mathtt{unit} \to \mathtt{unit}].\lambda x : \mathtt{var}[\mathbf{N}].M) \\ \mathtt{new}_{\mathtt{unit}\to\mathtt{unit}} \cdots \mathtt{new}_{\mathtt{unit}\to\mathtt{unit}}\ \mathtt{new}_{\mathbf{N}}.$$ □

Given this result, we can now construct a fully abstract model of $\mathcal{L}$ as follows. Define a preorder $\preccurlyeq$ on each hom-set of $\mathrm{Fam}(\mathcal{C})$ as follows: $f \preccurlyeq g : A \to B$ iff

$$\forall \alpha : A \Rightarrow B \to T\mathbf{1}.`f' ; \alpha \neq \bot \Rightarrow `g' ; \alpha \neq \bot,$$

where $`f' : \mathbf{1} \to (A \Rightarrow B)$ is obtained from f by currying. This preorder induces an equivalence relation on each hom-set in the usual way, and taking equivalence classes of maps yields another category $\mathrm{Fam}(\mathcal{C})/\preccurlyeq$. This category inherits all the relevant structure from $\mathrm{Fam}(\mathcal{C})$ and is therefore another sound model of $\mathcal{L}$. The above definability result implies that this model is fully abstract.

Theorem 8 (Full abstraction) The model of $\mathcal{L}$ in $\mathrm{Fam}(\mathcal{C})/\preccurlyeq$ is fully abstract.

Proof We just need to show completeness, and in fact we prove the contrapositive. Suppose M and N are two (wlog closed) terms of type A such that $[\![M]\!] \neq [\![N]\!]$. There must exist a strategy $\alpha : (\mathbf{1} \Rightarrow TA) \to T\mathbf{1}$ such that (wlog) $`[\![M]\!]' ; \alpha = \bot$ and $`[\![N]\!]' ; \alpha \neq \bot$. The map α can be taken to be a finite strategy, and the definability result implies that $\alpha = [\![x : \mathbf{1} \to A \vdash C[x]]\!]$ for some term $C[x]$. Also, $`[\![M]\!]' = [\![\lambda d.M]\!] ; \eta$ where d is a dummy variable; and similarly for N. It follows that $[\![C[\lambda d.M]]\!] = \bot$ and $[\![C[\lambda d.N]]\!] \neq \bot$, so by computational adequacy, $C[\lambda d.N]\Downarrow$ but not $C[\lambda d.M]\Downarrow$. □

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. Accepted for publication in Information and Computation, 1997.

[2] S. Abramsky and G. McCusker. Games and full abstraction for the lazy λ-calculus. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 234–243. IEEE Computer Society Press, 1995.

[3] S. Abramsky and G. McCusker. Call-by-value games. In *Proceedings of CSL '97*, Lecture Notes in Computer Science. Springer-Verlag, 1997. To appear.

[4] 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.

[5] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. To appear in Theoretical Computer Science, 1998.

[6] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus (extended abstract). In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science* [9], pages 43–54.

[7] 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.

[8] 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.

[9] IEEE Computer Society Press. *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, 1996.

[10] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.

[11] J. Laird. Full abstraction for functional languages with control. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 58–67. IEEE Computer Society Press, 1997.

[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* [9], pages 174–183.

[14] G. McCusker. Games and definability for FPC. *Bulletin of Symbolic Logic*, 3(3):347–362, Sept. 1997.

[15] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976.

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[18] R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

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

[20] 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.

[21] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science: Proceedings of the LMS Symposium, Durham, 1991*. Cambridge University Press, 1992. LMS Lecture Notes Series, 177.

[22] J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.

[23] J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[24] I. Stark. A fully abstract domain model for the π-calculus. In *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science* [9], pages 36–42.

[25] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, volume 1 of *The MIT Press Series in Computer Science*. The MIT Press, 1977.