URL: http://www.elsevier.nl/locate/entcs/volume3.html 13 pages

Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions Extended Abstract

Samson Abramsky [1]

Department of Computer Science
University of Edinburgh
King's Buildings
Mayfield Road
Edinburgh
United Kingdom EH9 3JZ.

Guy McCusker [2]

Department of Computing
Imperial College
180 Queen's Gate
London,
United Kingdom SW7 2BZ.

Abstract

The manipulation of objects with state which changes over time is all-pervasive in computing. Perhaps the simplest example of such objects are the program variables of classical imperative languages. An important strand of work within the study of such languages, pioneered by John Reynolds, focusses on "Idealized Algol", an elegant synthesis of imperative and functional features.

We present a novel semantics for Idealized Algol using games, which is quite unlike traditional denotational models of state. The model takes into account the irreversibility of changes in state, and makes explicit the difference between *copying* and *sharing* of entities. As a formal measure of the accuracy of our model, we obtain a full abstraction theorem for Idealized Algol with active expressions.

[1] Research supported by EPSRC grant "Foundational Structures for Computer Science".
[2] The second author is supported by an EPSRC Research Studentship.

1 Introduction

Our starting point is the elegant synthesis of functional and imperative programming due to John Reynolds [16]. His approach is to take an applied simply-typed λ-calculus in which, by suitable choices of the base types and constants, imperative features can be represented. The resulting language is referred to as *Idealized Algol* (IA for short).

The language is in fact parametrized by the choice of basic data types and operations on them. In Reynolds' approach, a sharp distinction is drawn between these basic data types—which can be regarded as discrete sets—and the phrase types of the programming language. For each basic data type = set X, there will be two basic phrase types of the language:

- $\mathtt{exp}[\mathtt{X}]$—the type of expressions which yield values in the set X.
- $\mathtt{var}[\mathtt{X}]$—the type of (assignable) program variables which can contain values from the set X.

In addition, there is a basic type $\mathtt{com}$ of commands. Reynolds makes a sharp distinction between expressions, which he does not allow to have side-effects, and commands, which are executed precisely to have side-effects. We shall relax this distinction, and can consequently identify $\mathtt{com}$ with $\mathtt{exp}[\mathtt{1}]$ where $\mathbf{1}$ is a one-point set—the "unit" type.

The syntax of basic types is then

$$B \quad ::= \quad \mathtt{exp}[\mathtt{X}] \mid \mathtt{var}[\mathtt{X}] \mid \mathtt{com}.$$

The general types are defined by

$$T \quad ::= \quad B \mid T \Rightarrow T.$$

The constants of the language will be as follows:

- Recursion combinators $\mathbf{Y}_T : (T \Rightarrow T) \Rightarrow T$ for each type T.
- Conditionals

  $$\mathtt{cond}_B : \mathtt{exp}[\mathbf{B}] \Rightarrow B \Rightarrow B \Rightarrow B$$

  for each base type B.
- For each operation

  $$f : X_1 \times \cdots \times X_k \longrightarrow X$$

  on the basic data types, a constant

  $$\tilde{f} : \mathtt{exp}[\mathtt{X_1}] \Rightarrow \cdots \mathtt{exp}[\mathtt{X_k}] \Rightarrow \mathtt{exp}[\mathtt{X}].$$
- Command sequencing:

  $$\mathtt{seq}_X : \mathtt{com} \Rightarrow \mathtt{exp}[\mathtt{X}] \Rightarrow \mathtt{exp}[\mathtt{X}]$$

  for each set X. Taking $X = \mathbf{1}$ this yields

  $$\mathtt{seq} : \mathtt{com} \Rightarrow \mathtt{com} \Rightarrow \mathtt{com}$$

  as a special case. Also,

  $$\mathtt{skip} : \mathtt{com}.$$
- Variable dereferencing and assignment:

  $$\mathtt{deref}_X : \mathtt{var}[\mathtt{X}] \Rightarrow \mathtt{exp}[\mathtt{X}]$$

$$\texttt{assign}_X : \texttt{var[X]} \Rightarrow \texttt{exp[X]} \Rightarrow \texttt{com}.$$

- Block structure:

$$\texttt{new}_{X,Y} : (\texttt{var[X]} \Rightarrow \texttt{exp[Y]}) \Rightarrow \texttt{exp[Y]}.$$

  Again, taking $Y = \mathbf{1}$ yields

$$\texttt{new}_X : (\texttt{var[X]} \Rightarrow \texttt{com}) \Rightarrow \texttt{com}$$

  as a special case.

The first three items mean that we can regard Idealized Algol as an *extension* of PCF [13], identifying $\texttt{exp}[\mathbf{N}]$, $\texttt{exp}[\mathbf{B}]$ with the basic types **nat**, **bool** of PCF. Reynolds' version of the language, in which expressions cannot have side-effects, is obtained by removing all constants $\texttt{seq}_X$, $\texttt{new}_{X,Y}$ except for $\texttt{seq}$, $\texttt{new}_X$.

With these constants, we can encode the usual imperative constructs.

Examples

(i) $x := x + 1$ is represented by

$$x : \texttt{var}[\mathbf{N}] \vdash \texttt{assign}\ x\ (\texttt{succ}\ (\texttt{deref}\ x)) : \texttt{com}.$$

(ii) $\texttt{while}\ \neg(x = 0)\ \texttt{do}\ x := x - 1$ is represented by

$$\begin{aligned} x : \texttt{var}[\mathbf{N}] \vdash \mathbf{Y}_{\texttt{com}}(\lambda c : \texttt{com}.\ \texttt{cond}\ (\texttt{iszero}\ x)& \\ &\texttt{skip} \\ &(\texttt{seq}\ (\texttt{assign}\ x\ (\texttt{pred}(\texttt{deref}\ x)))\ c)) : \texttt{com}. \end{aligned}$$

(iii) The code $\texttt{new}\ x\ \texttt{in}$

$$\begin{aligned} &(x := 0; \\ &p(x := x + 1); \\ &\texttt{if}\ x = 0\ \texttt{then}\ \Omega\ \texttt{else}\ \texttt{skip} \\ &) \end{aligned}$$

translates into

$$\begin{aligned} p : \texttt{com} \Rightarrow \texttt{com} \vdash\ &\texttt{new}(\lambda x : \texttt{var}[\mathbf{N}]. \\ &\quad \texttt{seq}\,(\texttt{assign}\ x\ 0) \\ &\quad\quad (\texttt{seq}\,(p(\texttt{assign}\ x\ (\texttt{succ}(\texttt{deref}\ x)))) \\ &\quad\quad\quad (\texttt{cond}\ (\texttt{iszero}\ (\texttt{deref}\ x))\ \Omega\ \texttt{skip}))) : \texttt{com}. \end{aligned}$$

It is worth noting how the translation into Idealized Algol forces us to be precise about the types of the variables, and whether they occur free or bound.

Idealized Algol is surprisingly expressive. For example, scoped arrays with dynamically computed bounds can be introduced by definitional extension [18], as can classes, objects and methods [15]. It can thus serve as a prototypical language combining state and block structure with higher-order functional features in the same way that PCF has been studied as a prototypical functional language.

2 Models for Idealized Algol

What should a model for Idealized Algol look like? Since the language is an applied simply typed λ-calculus, we should expect to model it in a cartesian-closed category $\mathcal{C}$ [5]. To accommodate the recursion in the language, we can ask for $\mathcal{C}$ to be cpo-enriched [3], or more minimally, to be *rational* [2]. This says that $\mathcal{C}$ is enriched over pointed posets, and that the least upper bounds of chains $\{f^k \circ \bot \mid k \in \omega\}$ of iterated endomorphisms $f : A \rightarrow A$ exist in a suitably enriched and parametrized sense (see [2] for details).

As regards the basic data types, following [1] we can ask that the functor

$$\mathcal{C}(1, -) : \mathcal{C}_0 \longrightarrow \mathbf{Set}_\star$$

have a left adjoint left inverse $\widetilde{(\cdot)}$, where $\mathcal{C}_0$ is a suitable sub-category of $\mathcal{C}$, and $\mathbf{Set}_\star$ is the category of pointed sets. (If $\mathcal{C}$ is $\mathbf{Cpo}$, then $\mathcal{C}_0$ will be the sub-category of strict maps, and $\widetilde{X_\star}$ is the flat domain $X_\bot$. If $\mathcal{C}$ is a category of games, then $\mathcal{C}_0$ will be the sub-category of total morphisms on π-atomic objects, and $\widetilde{X_\star}$ will be the flat game over the set X; see [1].)

Unpacking this definition, it says that for each object A of $\mathcal{C}_0$ and pointed set $X_\star$ (where X is the set of elements of $X_\star$ excluding the basepoint), there is a bijection

$$\mathcal{C}_0(\widetilde{X_\star}, A) \cong \mathbf{Set}_\star(X_\star, \mathcal{C}(1, A))$$

natural in $X_\star$ and A, and such that the unit

$$\eta_{X_\star} : X_\star \longrightarrow \mathcal{C}(1, \widetilde{X_\star})$$

is a bijection. This means that each $x \in X$ corresponds to a unique morphism $\bar{x} : 1 \rightarrow \widetilde{X_\star}$ in $\mathcal{C}$, and for each family

$$(f_x : 1 \longrightarrow A \mid x \in X)$$

there is a unique $\mathcal{C}_0$-morphism

$$[f_x \mid x \in X] : \widetilde{X_\star} \longrightarrow A$$

such that

$$\overline{x_0}; [f_x \mid x \in X] = f_{x_0}$$

for all $x_0 \in X$. As a special case, each $f : X_\star \rightarrow Y_\star$ in $\mathbf{Set}_\star$ (*i.e.* each partial function $f : X \rightharpoonup Y$) is represented by

$$[g_x \mid x \in X] : \widetilde{X_\star} \longrightarrow \widetilde{Y_\star}$$

where

$$g_x = \begin{cases} \overline{f(x)}, & f(x) \text{ defined} \\ \bot_{\widetilde{Y_\star}}, & \text{otherwise.} \end{cases}$$

Thus this adjunction gives us a "constant" or "numeral" for each of the elements of a datatype, and a `case` construct to branch on the values of a datatype, with (partial) functions between datatypes represented as special cases.

This data is used in modelling the basic types and constants of Idealized Algol as follows:

- $\mathtt{exp[X]}$ is interpreted by $\widetilde{X_\star}$, and $\mathtt{com} = \mathtt{exp[1]}$ by $\widetilde{1_\star}$.
- The conditionals and basic operations on data types are interpreted using the `case` construct. For example, the constant

  $$\mathtt{if0} : \mathtt{exp}[\mathbf{N}] \Rightarrow \mathtt{exp}[\mathbf{N}] \Rightarrow \mathtt{exp}[\mathbf{N}] \Rightarrow \mathtt{exp}[\mathbf{N}]$$

  is represented by $[g_n \mid n \in \mathbf{N}]$, where

  $$g_0 = [\![\lambda x, y : \mathtt{exp}[\mathbf{N}].\, x]\!]$$
  $$g_{n+1} = [\![\lambda x, y : \mathtt{exp}[\mathbf{N}].\, y]\!].$$

- Skip and sequencing are represented as the degenerate cases of constants and case constructs for the underlying data type $X = \mathbf{1} = \{\bullet\}$. That is,

  $$[\![\mathtt{skip}]\!] = \top \overset{\mathrm{df}}{=} \bar{\bullet} : \mathbf{1} \longrightarrow \widetilde{1_\star}$$
  $$[\![\mathtt{seq}_A]\!] = [g_\bullet \mid \bullet \in \{\bullet\}] : \widetilde{1_\star} \Rightarrow A \Rightarrow A$$

  where $g_\bullet = [\![\lambda x : A.\, x]\!]$.

  To understand this modelling of sequencing, think of $\widetilde{1_\star}$ as a type with a single defined value (the denotation $\top$ of `skip`); this value being returned when a command is invoked corresponds to the successful termination of the command. Now just as a conditional firstly evaluates the test, and then selects one of the branches for execution, so `seq` as a "unary case statement" will firstly evaluate the command, and then, provided the command terminates successfully, proceed with the continuation.

What about variables? Here we can make use of another important idea due to Reynolds [16]. If we analyze the "methods" we use to manipulate variables, in the spirit of object-oriented programming, we see that there are just two:

- the reading (or dereferencing) method: $\mathtt{deref}_X : \mathtt{var[X]} \Rightarrow \mathtt{exp[X]}$.
- the writing (or assignment) method: $\mathtt{assign}_X : \mathtt{var[X]} \Rightarrow \mathtt{exp[X]} \Rightarrow \mathtt{com}$.

We can *identify* $\mathtt{var[X]}$ with the product of its methods:

$$\mathtt{var[X]} = \mathtt{acc[X]} \times \mathtt{exp[X]}$$

where $\mathtt{acc[X]}$ is the type of "acceptors" or write capabilities. It is then tempting to take a further step (which Reynolds has often, but not always, taken [16]; *cf.* also [18]), and identify acceptors with the type indicated by the assignment operation:

$$\mathtt{acc[X]} = \mathtt{exp[X]} \Rightarrow \mathtt{com}.$$

However, we do not wish to do this, for the following reason. Idealized Algol is a call-by-name language, but it is intended that only *values* of basic data types, not "closures" or "thunks", be stored in program variables. Thus we prefer to define

$$\mathtt{acc[X]} = \mathtt{com}^X,$$

the product of X copies of `com`. We can then define a *retraction*

$$c : \mathtt{acc[X]} \lhd (\mathtt{exp[X]} \Rightarrow \mathtt{com}) : r$$

where $c : \mathtt{acc[X]} \to (\mathtt{exp[X]} \Rightarrow \mathtt{com})$ is defined by exponential transposition from

$$\hat{c} : \widetilde{X_\star} \longrightarrow (\mathtt{com}^X \Rightarrow \mathtt{com})$$

$$\hat{c} = [\pi_x \mid x \in X],$$

and

$$r = \langle r_x \mid x \in X\rangle : (\mathtt{exp[X]} \Rightarrow \mathtt{com}) \longrightarrow \mathtt{acc[X]}$$

where $r_x : (\mathtt{exp[X]} \Rightarrow \mathtt{com}) \to \mathtt{com}$ is defined by:

$$\mathtt{exp[X]} \Rightarrow \mathtt{com} \cong (\widetilde{X_\star} \Rightarrow \mathtt{com}) \times 1 \xrightarrow{\mathtt{id}\times\bar{x}} (\widetilde{X_\star} \Rightarrow \mathtt{com}) \times \widetilde{X_\star} \xrightarrow{\mathtt{Ap}} \mathtt{com}.$$

If these definitions are worked out in the case of **Cpo**, it will be seen that r is "strictification", while c is just inclusion of strict functions, as expected.

Given these definitions, we can now define

$$[\![\mathtt{deref}]\!] = \mathtt{snd}$$

$$[\![\mathtt{assign}]\!] = c \circ \mathtt{fst}.$$

Since we have omitted products from our version of Idealized Algol for notational simplicity, it also makes sense to define a variable constructor

$$\mathtt{mkvar}_X : (\mathtt{exp[X]} \Rightarrow \mathtt{com}) \Rightarrow \mathtt{exp[X]} \Rightarrow \mathtt{var[X]}$$

by

$$[\![\mathtt{mkvar}_X]\!] fe = \langle r \circ f, e\rangle.$$

3 The Functional/Imperative Boundary

At this point, the reader should be experiencing a sense of vertigo, or at least puzzlement. We have provided a notion of model for Idealized Algol which is only the mildest extension of the usual notion of model for PCF, and yet which appears to account for all the imperative features of the language, without introducing states or any other device for explicitly modelling assignable variables! What is going on?

The answer is indeed a very interesting consequence of Reynolds' analysis of imperative languages, although it is one which, as far as we are aware, he has not himself explicitly drawn. Firstly, note that a more precise statement is that the notion of model we have developed to this point accounts for everything in Idealized Algol *except* the `new` constants, *i.e.* block structure. We refer to the sub-language obtained by omitting the `new` constants as $\mathsf{IA} - \{\mathtt{new}\}$. We can now formulate the following thesis:

$$\boxed{\mathsf{IA} - \{\mathtt{new}\} \text{ is a pure functional language.}}$$

At first sight, this seems nonsensical, since the usual "basic imperative language" [19], which does not include block structure, can be represented in $\mathsf{IA} - \{\mathtt{new}\}$, as shown in Section 1. However, recall that the process of translating an imperative language into IA forced us to be more explicit about free and bound variables. The "basic imperative language" of the textbooks actually relies on an implicit convention by which the program variables, which are all global, are bound (and possibly initialized) at the top level. We claim that it is only when identifiers of type `var[X]` are bound to actual "storage cell objects"—which is exactly what the `new` constants do—that real imperative behaviour arises.

Of course, to substantiate this claim, we must show, not only that our simple specification of a "functional model" for $\mathsf{IA} - \{\texttt{new}\}$ suffices to interpret the syntax, but that actual models so arising do faithfully reflect the concepts in the language, and capture the operational behaviour of programs. We can in fact do this in a very strong sense. As we shall see in Section 6, the categories of games used to give the first syntax-independent constructions of fully abstract models for PCF [2,6], when used to give models for $\mathsf{IA} - \{\texttt{new}\}$ in the way we have described, again yield fully abstract models. Moreover, the *proof* of full abstraction is a very easy extension of that for PCF, and can be given at the axiomatic level introduced in [1]. This latter point means that *any* model of the axioms in [1] yields a fully abstract model of $\mathsf{IA} - \{\texttt{new}\}$.

Firstly, however, we shall turn to the question of modelling the `new` constants.

4 The semantics of `new`

Our previous discussion has located the functional/imperative boundary, the point at which genuinely "stateful" behaviour arises, in the semantics of the `new` constant. What are the key features of this construct?

Locality The "object" created by a local declaration

$\texttt{new}\ x\ \texttt{in}\ C$

must be "private" to C. This causes problems for traditional models based on representing the state in a global, monolithic fashion by a mapping from "locations" to values. The functor-category approaches [12,18] address this problem by replacing the global state by a functor varying over "stages".

Irreversibility When a variable is updated, the previous value is lost. Again, models based on representing states as functions find it hard to account for this feature. For good discussions of this point see [11,14].

Sharing Multiple occurrences of a variable in a functional program refer, conceptually at least, to different "copies" of the same, unchanging value ("referential transparency"); this implies that the temporal order in which these occurrences are dereferenced makes no difference to the outcome. By contrast, multiple references to an assignable variable refer to different time-slots in the life of a single underlying object with state which changes over time; this is *sharing* rather than *copying*.

How can we capture these features? The point of view we wish to adopt is one we have already hinted at, and indeed appears in a significant line of previous work [9,10,14]. We want to understand $\texttt{new}\ x\ \texttt{in}\ C$ as binding the free identifier x of type var[X] to an "object" or "process" which gives the behaviour of a storage cell. The behaviour of `new` x `in` C then arises from the *interaction* between C and this cell, which is "internalized", *i.e.* hidden from the environment. Such an account immediately addresses two of the key features of `new` noted above:

- Locality is addressed, since the interaction between C and the storage cell process is hidden from the environment.
- Irreversibility is addressed, since the state of the storage cell will change as C interacts with it.

How can we formalize this idea in our current framework? A first attempt is to consider introducing a constant

$$\mathtt{cell}_X : 1 \longrightarrow \mathtt{var[X]}$$

such that, if $f : \mathtt{var[X]} \longrightarrow A$, $\mathtt{new}(f)$ is given by the composition

$$\mathtt{new}(f) = 1 \xrightarrow{\mathtt{cell}_X} \mathtt{var[X]} \xrightarrow{f} A.$$

The idea is that $\mathtt{cell}_X$ gives the "behaviour" of our storage cell process. However, recalling that

$$\mathtt{var[X]} = \mathtt{com}^X \times \mathtt{exp[X]}$$

this is clearly hopeless, since a constant of this type, which in particular will supply a constant value every time we read from the variable, is clearly just what we *don't* want!—We need to take account of the changing state of the variable.

At this point we produce our *deus ex machina*: Linear Logic! Up to this point, we have been working exclusively with intuitionistic types; since everything except `new` was essentially functional, this was all we needed, at least to get a model. But now we need a loop-hole to get some access to the dynamics, and Linear Logic provides such a loop-hole. Suppose then that our cartesian closed category $\mathcal{C}$ arises as $\mathcal{C} = K_!(\mathcal{L})$, the co-Kleisli category of a Linear category $\mathcal{L}$ with respect to the ! comonad [4,17]. The intuitionistic function types we have been using get their standard decompositions into the Linear types:

$$A \Rightarrow B = !A \multimap B.$$

In particular, we see that the type of $\mathtt{new}_A$ is:

$$\mathtt{new}_{X,A} : !(!\mathtt{var[X]} \multimap \mathtt{exp[A]}) \multimap \mathtt{exp[A]}.$$

Now suppose that we have a morphism

$$\mathtt{cell}_X : I \longrightarrow !\mathtt{var[X]}.$$

Then we can define $\mathtt{new}_A$ as the composition

$$\begin{tikzcd}
!(\,!\mathtt{var[X]} \multimap \mathtt{exp[A]}) \arrow[d, "\mathtt{der}"] \\
!\mathtt{var[X]} \multimap \mathtt{exp[A]} \arrow[d, "\cong"] \\
(\,!\mathtt{var[X]} \multimap \mathtt{exp[A]}) \otimes I \arrow[d, "\mathtt{id} \otimes \mathtt{cell}_X"] \\
(\,!\mathtt{var[X]} \multimap \mathtt{exp[A]}) \otimes !\mathtt{var[X]} \arrow[d, "\mathtt{Ap}"] \\
\mathtt{exp[A]}.
\end{tikzcd}$$

(Here `der` is the dereliction map (the counit of !), and `Ap` is the linear application.)

Note that Linear types really are necessary here. If we had a constant $\mathtt{cell}_X : \mathtt{var[X]}$ in the language, and tried to define

$$\mathtt{new} f = f \mathtt{cell}_X,$$

then this would be interpreted in $K_!(\mathcal{L})$ by

$$I \xrightarrow{\mathtt{cell}_0^\dagger} !\mathtt{var[X]} \xrightarrow{f} A$$

where $\mathtt{cell}_0^\dagger$ is the "promotion" of a morphism

$$\mathtt{cell}_0 : I \longrightarrow \mathtt{var[X]}.$$

But the promotion will behave "uniformly" in each copy of $\mathtt{var[X]}$, whereas we clearly need behaviour which is history-sensitive, and depends on the previous history of accesses to other "copies" (which are really the previous time slots of the single shared underlying object with state). Thus the `cell` morphism we require will *not* be of the form $\mathtt{cell}_0^\dagger$ for any $\mathtt{cell}_0 : I \to \mathtt{var[X]}$.

Provided that we can define a suitable morphism

$$\mathtt{cell}_X : I \longrightarrow !\mathtt{var[X]}$$

which does capture the behaviour of a storage cell object, then we have completed our semantics of Idealized Algol. In Section 7 we shall see that this can indeed be done for a suitable category of games, and by this means we will obtain the first fully abstract model of Idealized Algol.

The point to be emphasized here is how small an increment from the modelling of PCF is required to capture Idealized Algol, *provided* a sufficient handle on the dynamics is present in our semantics in order to define the `cell` morphism. The key feature of game semantics is that *the dynamics is already there.*

How is sharing represented in this approach? Firstly, the multiple references to a variable are interpreted using the cocommutative comonoid structure of $!\mathtt{var}[\mathtt{X}]$, *i.e.* the contraction rule, so that the interpretation of a block $\mathtt{new}\ x\ \mathtt{in}\ C$ looks like:

$$\begin{tikzcd}
 & & !\mathtt{var}[\mathtt{X}] & \\
 & & \otimes & \\
I \arrow[r, "\mathtt{cell}"] & !\mathtt{var}[\mathtt{X}] \arrow[uur] \arrow[ddr] & \vdots \arrow[r, "{[\![C]\!]}"] & \mathtt{com} \\
 & & \otimes & \\
 & & !\mathtt{var}[\mathtt{X}] &
\end{tikzcd}$$

The contraction merges the accesses to the variable x arising from the various occurrences of it in C into a single "event stream". The task of the `cell` morphism is to impose the appropriate causality on this event stream, so that in particular a read will return the last value written.

In this extended abstract, we will only outline the contents of the remaining sections of the full paper, which will flesh out the technical details of our approach.

5 Games

We describe a number of categories of games: linear categories $\mathcal{G}$ and $\mathcal{G}_{\mathrm{inn}}$, and cartesian closed categories $\mathcal{C}$ and $\mathcal{C}_{\mathrm{inn}}$. $\mathcal{G}_{\mathrm{inn}}$ and $\mathcal{C}_{\mathrm{inn}}$ are the categories of games and innocent strategies introduced in [7,8]. They are subcategories of $\mathcal{G}$ and $\mathcal{C}$ respectively, which arise by dropping the innocence constraint on strategies; we say that non-innocent strategies are *knowing* for emphasis and contrast.

The representation of basic data types in $\mathcal{G}_{\mathrm{inn}}$ as flat games is described, and the interpretation of the basic types and constants of $\mathsf{IA} - \{\mathtt{new}\}$ in $\mathcal{G}_{\mathrm{inn}}$ according to the general scheme of Section 2 is spelled out.

6 $\mathsf{IA} - \{\mathtt{new}\}$

The following result is proved.

Theorem 6.1 (Innocent Definability) *Let $\mathcal{K}$ be a sequential category in the sense of [1], and T an* IA *type. Then any compact morphism*

$$f : 1 \longrightarrow [\![T]\!]$$

in $\mathcal{K}$ is definable in $\mathsf{IA} - \{\mathtt{new}\}$. In particular, this holds for $\mathcal{C}_{\mathrm{inn}}$.

The proof is a straightforward extension of that in [1], and yields a normal form for $\mathsf{IA} - \{\mathtt{new}\}$ extending the evaluation trees of [2].

7 Modelling `new`

The interpretation of $\mathtt{cell}_X : I \longrightarrow !\mathtt{var}[\mathtt{X}]$ as a *non-innocent* strategy in $\mathcal{G}$ is specified, and also "initialized cells" $\mathtt{cell}_{X,x_0}$ for each $x_0 \in X$. Since the denotations of all terms in $\mathsf{IA} - \{\mathtt{new}\}$ are innocent strategies, this raises the question of which sub-category of $\mathcal{G}$ is generated by $\mathcal{G}_{\mathrm{inn}} + \{\mathtt{cell}\}$. The answer is provided by the following result.

Theorem 7.1 (Innocent Factorization) *Let A be a game such that*

$$sa \in P_A^{\mathrm{odd}} \wedge \ulcorner sa \urcorner b \in P_A \implies sab \in P_A,$$

and $\sigma : I \longrightarrow A$ be any (knowing) strategy. Then there exists a set X, $x_0 \in X$, and an innocent strategy $\tau : !\mathtt{var}[\mathtt{X}] \longrightarrow A$ such that

$$\sigma = \mathtt{cell}_{X,x_0} ; \tau.$$

Moreover, if σ is compact, so is τ.

The proof of this weak orthogonality property is by a simulation argument; the strategy τ uses the variable—which it interacts with innocently—to encode the full history of the play in A.

For any IA type T, the game $[\![T]\!]$ satisfies the hypothesis of the Innocent Factorization Theorem. Combining this with Innocent Definability, we therefore obtain:

Theorem 7.2 (Definability) *Every compact strategy $\sigma : I \longrightarrow [\![T]\!]$ in $\mathcal{C}$, where T is an IA type, is definable in Idealized Algol.*

8 Computational Adequacy

The operational semantics of Idealized Algol is defined, and computational adequacy is proved.

9 Full Abstraction

The following result is an easy consequence of Computational Adequacy and Definability.

Theorem 9.1 (Full Abstraction) *$\mathcal{C}/\lesssim$ is a fully abstract model of Idealized Algol, where $\lesssim$ is the intrinsic preorder on $\mathcal{C}$ [2,6,1].*

10 The Extensional Model

An explicit characterization of the intrinsic preorder on $\mathcal{C}$ is proved.

Theorem 10.1

$$\sigma \lesssim \tau \Longleftrightarrow \mathsf{comp}(\sigma) \subseteq \mathsf{comp}(\tau)$$

$$\iff \mathsf{Pref}(\mathsf{comp}(\sigma)) \subseteq \mathsf{Pref}(\mathsf{comp}(\tau))$$

where $\mathsf{comp}(\sigma)$ *is the set of "complete plays" of* σ*, i.e. maximal plays in which all questions have been answered.*

As an easy consequence of this result, we obtain:

Theorem 10.2 (Effective Presentability) *The fully abstract model of Idealized Algol is effectively presentable.*

It is still unknown if the analogous result holds for PCF.

References

[1] S. Abramsky. Axioms for full abstraction and full completeness. Submitted for publication, 1996.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. Submitted for publication, 1996.

[3] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Semantics*, pages 89–132. Cambridge University Press, 1986.

[4] G. Bierman. What is a categorical model of intuitionistic linear logic? In *International Conference onTyped Lambda Calculi and Applications*. Springer-Verlag, 1995. Lecture Notes in Computer Science.

[5] R. Crole. *Categories for Types*. Cambridge University Press, 1994.

[6] M. Hyland and C.H. L. Ong. On full abstraction for PCF. Submitted for publication, 1996.

[7] G. McCusker. *Games and Full Abstraction for a functional metalanguage with recursive types*. PhD thesis, Imperial College, University of London, 1996. to appear.

[8] G. McCusker. Games and full abstraction for FPC. In *International Symposium on Logic in Computer Science*, 1996.

[9] R. Milner. Processes: a mathematical model of computing agents. In *Logic Colloquium '73*, pages 157–173. North Holland, 1975.

[10] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[11] P. W. O'Hearn and U. Reddy. Objects, interference and the Yoneda embedding. In M. Main and S. Brookes, editors, *Mathematical Foundations of Programming Semantics: Proceedings of 11th International Conference*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B.V., 1995.

[12] F. J. Oles. Type categories, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Semantics*. Cambridge University Press, 1985.

[13] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[14] U. Reddy. Global state considered unncessary: an object-based semantics for Algol. *Lisp and Functional Programming*, 1996.

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

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

[17] R. A. G. Seeley. Linear logic, $*$-autonomous categories and cofree coalgebras. In *Category theory, computer science and logic*. American Mathematical Society, 1987.

[18] R. D. Tennent. Denotational semantics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 169–322. Oxford University Press, 1994.

[19] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, 1993.