

# Objects, Interference, and the Yoneda Embedding

Peter W. O'Hearn<sup>1</sup>

*Syracuse University*

Uday S. Reddy<sup>2</sup>

*University of Illinois at Urbana-Champaign*

*Dedicated to John C. Reynolds, in honor of his 60th birthday.*

---

## Abstract

We present a new semantics for Algol-like languages that combines methods from two prior lines of development:

- the object-based approach of [21,22], where the meaning of an imperative program is described in terms of sequences of observable actions, and
- the functor-category approach initiated by Reynolds [24], where the varying nature of the run-time stack is explained using functors from a category of store shapes to a category of cpos.

The semantics gives an account of both the phenomena of local state and irreversibility of state change. As an indication of the accuracy obtained, we present a full abstraction result for closed terms of second-order type in a language containing active expressions, i.e. value-returning commands.

---

## 1 Introduction

In his influential Turing award lecture [1], John Backus criticized imperative programming languages for promoting a view of programming as “word-at-a-time” processing. John Reynolds expressed his response to this criticism in a meeting of IFIP working group 2.2 in around 1988 (which he repeated to several people privately, including the second author). The view put forward by Backus, Reynolds said, is that imperative programming is like working with

---

<sup>1</sup> Supported by NSF grant CCR-92-110829.

<sup>2</sup> Supported by NSF grant CCR-93-03043.

“pigeon holes.” All that one does is to take a pigeon out from a hole or to put a new pigeon in a hole. But, with object-oriented programming, he said, one works with “turkey holes” rather than pigeon holes. Instead of taking out a pigeon or putting in a pigeon, one does more sophisticated manipulations such as “rotate a turkey” or “tilt a turkey.”

The “turkey holes” that Reynolds spoke of are what programmers call “objects”. They occupy some physical space, whose contents can be altered, and support operations for the manipulation of these contents. Programs are built by putting such objects together and letting them act on each other. Such an object-based view, we find, is implicit throughout Reynolds’s work on imperative programming.

In his seminal paper [24] on Algol-like languages, Reynolds treats procedures, not as actions on the global state, but as actions on the state at the point of their definitions. Every procedure lives in its own “turkey hole,” so to speak. Reynolds also shows how to treat variables (“pigeon holes”) as a special case of turkey holes — objects with operations for setting and reading values stored in them. This essentially frees imperative programming from the limitations suggested by Backus and sets up a truly object-based paradigm for thinking about imperative programs.

Reynolds’s program for the semantics of imperative languages was further developed by Oles and Tennent [16,17,29–31], and continued and expanded in a number of works [9,10,18,12,14,27,11,26]. In a separate line of development, a model based more explicitly on a notion of “objects” has been formulated in [21,22]. Reynolds’s conception of imperative programming expressed above formed an important pre-theoretic motivation for this work, though its theoretical development also draws inspiration from linear logic, syntactic control of interference, and the relation between them. In this paper we obtain a new semantics for Algol-like languages via a synthesis of these two lines,

- the “object-based” approach of [21], where the meaning of an imperative program is described in terms of sequences of observable actions, and
- the functor-category approach initiated by Reynolds [24], where the varying nature of the run-time stack is explained using functors from a category of store shapes to a category of cpos.

In the remainder of this introductory section we give an informal overview of the construction and discuss the specific semantic issues addressed by it.

### *1.1 Semantic issues: Locality and irreversibility*

In imperative computation there is an idea of destroying information by overwriting parts of computer memory. This is clearly important for implementation. But supplying direct access to assignment in the programming language also results in positive information that programmers make use of. Consider a parameterless procedure **gensym** that returns a different integer each time it is called. In reasoning about a program using **gensym**, for instance generating fresh names when implementing substitution in  $\lambda$ -calculus, we would

use the property that any call to **gensym** returns an integer that was not returned by it previously.

This property exemplifies one of the most basic intuitions about state: the (general) irreversibility of state change. By this we mean not only that portions of the store are destructively updated during the course of a computation, but that in the presence of abstraction or local state this irreversibility manifests itself in observable properties of programs.

A typical implementation of **gensym** would use a local integer variable that is incremented on each call. When we say that **gensym** returns a different integer each time it is called, it is crucial that *other* procedures or objects do not access the local state of **gensym** directly, and reset the value to a previously-encountered one. This statement about the **gensym** procedure implicitly involves interactions between the procedure and any other pieces of a program. The following code illustrates the kind of property of such interactions we have in mind.

```

begin
  integer  $x$ ;
  integer procedure gensym; {  $x := x + 1$ ; return( $x$ ); }
   $x := 0$ ;
   $P(\mathbf{gensym})$ ;
  if ( $\mathbf{gensym} > 1$ ) then diverge
end

```

A “client” procedure  $P$  is passed a parameterless procedure, **gensym**, for generating new names.  $P$  can use its argument a number of times (we are assuming call-by-name, though the effect can obviously be simulated in call-by-value), and if it uses its argument at least once then we expect that the whole block will diverge. Since the non-local procedure  $P$  cannot access the local variable  $x$ , if  $x$  is updated by calling **gensym** then procedure  $P$  has no way of resetting its value to zero. It follows (by intuitive reasoning) that this block should have termination/non-termination behavior equivalent to  $P(\mathbf{diverge})$ .

This code is not a realistic program, but it is interesting for the reasoning principle it illustrates. Generally, when we have an object consisting of some internal state and observable operations, it is not possible for a client program to cause the internal state of the object to backtrack to previous states. This is because the only changes to the internal state that the client can possibly effect come about by using the provided operations. The (observable) ramifications of irreversibility of state change are inextricably bound up with locality.

Irreversibility has proven difficult to capture in semantics because most models allow for “snapback” operations. These operations work by accepting a procedure as an argument, running the procedure, and then restoring the state to the value it had before the argument was executed (this would contradict the reasoning about **gensym** above). The snapback effect requires restoration of even local state.

The phenomenon of irreversibility is not so clear cut in languages that

violate the abstractness of local state (such as  $C$ ), or when programming on a “system level” where one might want access to the entire computer memory. One could in some instances achieve the effect of snapback by a series of incremental state changes. But on the level of programmable objects where abstraction is central, irreversibility is a familiar phenomenon, one that arises in Scheme, ML, Algol, and most object-oriented languages.

These intertwined notions of irreversibility and locality are fundamental, and should be accounted for by a satisfactory theory of state.

### 1.2 Overview of Approach: Objects plus Yoneda

The model presented here builds upon the work reported in [21,22], where a semantics is presented based on identifying an imperative computation with a stream of observations. For example, commands are modelled not as state-to-state functions, but as sequences of signals ‘\*’ indicating a message to a “command object.” More accurately, a command-in-context  $\Gamma \vdash C : \mathbf{comm}$  translates demands for output, \*, into requests of  $\Gamma$ -typed entities. Similarly, “active integers” are modelled using streams of integers, where we read a stream  $\langle 3, 4 \rangle$  as indicating an object with a single operation that returns 3 the first time it is used, and 4 the second.

There is a view of an active integer as an object possessing an internal state that may change, and a method for accessing this state. But the representation of the state is finessed in the mathematical description of objects given in [22]; state is regarded as implicit in a history of events. One benefit of such a “stateless” account of state is that it forces locality to be respected when composing meanings. Since the internal state of an object is not part of the mathematical description, the ways of combining these entities does not “tamper” with the internal state in the way that early denotational models do [9]. Also, there is no explicit state to be subject to a snapback effect, though care is needed to compose meanings in a way that respects some temporal ordering.

The work reported in [21,22] formalizes these ideas, and results in a model that accounts for locality and irreversibility quite well. But there is one difficulty: in the treatment of state as an implicit attribute, it is not easy to give a satisfactory account of shared state. Put another way, the objects of [21,22] are non-interfering, and it is not obvious how to deal smoothly with interference. A notion of function type is defined, but it forms a monoidal closed structure obtained as the adjoint of a “non-interfering” (and non-Cartesian) product  $\otimes$  whose components do not interfere:  $(A \otimes B, C) \cong (A, B \rightarrow C)$ . As a result, the semantics is defined only for “syntactic control of interference,” a restricted form of  $\lambda$ -calculus [23,13]. The constraints in this framework disallow interference between procedure and argument (or client and object).

In order to treat the full (typed)  $\lambda$ -calculus, a semantics is called for based on a Cartesian closed category. The approach that we use here is brutally straightforward. We begin with a category  $\mathbf{C}$  of “object spaces” suitable for the semantics from [22], and simply apply a Yoneda embedding  $\mathbf{C} \rightarrow \mathbf{Cpo}^{\mathbf{C}^{\text{op}}}$

that maps this object-based semantics into a Cartesian closed category of (certain) functors, where  $\mathbf{Cpo}$  is the category of  $\omega$ -complete pointed posets and continuous functions. So, for instance, where the type of commands is interpreted as an object  $comm$  of category  $\mathbf{C}$ , in the functor category it is interpreted as the functor  $\mathbf{C}(-, comm)$  (using the order structure of  $\mathbf{C}$ ). Interpretations of first-order constants are obtained immediately, using the morphism part of the embedding functor and the fact that Yoneda preserves products. This much is rather obvious.

What is less obvious is how to interpret variable declarations, which semantically correspond to a second-order operation, and so are not immediately given by the Yoneda embedding. This is where Reynolds’s insight on local state and functor categories enters the picture [24]. We now view objects of  $\mathbf{C}$  as possible worlds describing “contexts of evaluation.” The meaning of a command  $\mathbf{new} x. C$  at possible world  $X$  is given in terms of the meaning of  $C$  in an enlarged context  $X \times var$ , where  $var$  is a space interpreting a storage variable.

What is finally least obvious, and perhaps surprising, is that this simple-minded approach using the Yoneda embedding should yield a very good model, and not only a barely adequate one (in the technical sense). There is a more general question of what properties of  $\mathbf{C}$  are needed to give a good model of state in  $\mathbf{Cpo}^{\mathbf{C}^{op}}$ , and we do not have an answer to this question at present. But for the specific  $\mathbf{C}$  that we consider, we give two main technical results that are an indication of the accuracy obtained.

- We give explicit representations of first-order types, and show that all natural transformations between (products of) base types are least upper bounds of definable elements. The language used for definability is an Algol-like language containing “active expressions,” i.e value-returning commands.
- We give a full abstraction result for closed terms of second-order type.

It is natural to ask whether we could obtain similar results without passing to a functor category, by expressing the ideas of [22] directly in a CCC obtained, perhaps, by leaving the framework of coherent spaces. This might be possible if we were to take a concurrent view of objects and accept non-determinism, but the details of such a treatment are by no means obvious. As we explain in section 4, the Yoneda interpretation accounts for interference via a determinate use of interleaving in which interfering objects are interpreted in a shared “context of evaluation.”

## 2 Syntax

We consider a language with the following base types:

- **comm**, the type of commands, and
- **aint**, the type of active integer expressions (“active integers,” for short).

By active expressions, we mean computations that (potentially) cause state changes and return values. We form other types using binary product  $\times$  and

function space  $\rightarrow$ . We follow Reynolds and regard a type **var** of storage variables as sugar for  $(\mathbf{aint} \rightarrow \mathbf{comm}) \times \mathbf{aint}$ . De-referencing is second projection, and assignment is accomplished with the first projection and procedure call. For instance,  $x := x + 1$  desugars as  $\pi_1(x)(\mathbf{succ}(\pi_2(x)))$ .

The type system is that of simply-typed  $\lambda$ -calculus, with binary products. The constants are as follows:

$$\begin{array}{ll} \mathbf{succ}, \mathbf{pred} : \mathbf{aint} \rightarrow \mathbf{aint} & \mathbf{Y}_t : (t \rightarrow t) \rightarrow t \\ \mathbf{ifz} : \mathbf{aint} \times b \times b \rightarrow b & 0 : \mathbf{aint} \\ \mathbf{new} : (\mathbf{var} \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} & \mathbf{skip} : \mathbf{comm} \\ \mathbf{letval} : b \rightarrow (b \rightarrow b') \rightarrow b' \end{array}$$

where  $b, b'$  range over base types and  $t$  over types.

The arithmetic constants are just those of sequential PCF. For commands, we have constants for local creation and a form of sequential composition **letval**. The phrase **letval**  $M (\lambda y N)$  evaluates  $M$ , binds the value obtained to  $y$ , and then executes  $N$ . In case  $M$  is a command,  $y$  is bound to **skip** after the execution of  $M$ . The key point here is that the execution of  $M$  can change the state, but subsequent uses of  $y$  do not. Also, the side-effect of  $M$  is persistent, and not a snapback. We use notation  $C; C'$  as sugar for **letval**  $C (\lambda x.C')$  where  $x$  not free in  $C'$  or  $C$ . this is for any combination of base types for  $C$  and  $C'$ . When  $C$  is a command and  $C'$  an integer, this gives us a side-effecting, or “active,” integer.

In our very bare sample language there is no input/output or global variables for programs to act upon. Storage variables are created using **new**, as in **new**  $(\lambda x.C)$ . This creates a local variable  $x$  (initialized to 0) that may be updated within  $C$  (recall the sugaring of assignment above), but this storage variable is de-allocated on block exit. As a result, a closed term of type **comm** does not change the state at all: it must be equivalent to  $\mathbf{Y}(\lambda x.x)$  or **skip**. But, even for this bare language, there are many interesting examples that illustrate principles of imperative computation [9,15].

**Example 2.1** The gensym example from the Introduction is represented as the following term.

$$\begin{array}{l} \mathbf{new} \lambda x. \\ \quad (\lambda \mathit{gensym} \\ \quad \quad \mathbf{letval} \pi_1(x) 0 \lambda dd. \\ \quad \quad \mathbf{letval} P(\mathit{gensym}) \lambda dd'. \\ \quad \quad \mathbf{ifz}(\mathbf{pred}(\mathit{gensym}), \mathbf{skip}, \mathbf{Y}(\lambda x.x)) \\ \quad \quad (\mathbf{letval}(\pi_1(x)(\mathbf{succ}(\pi_2(x))) \lambda y.\pi_2(x))) \end{array}$$

where  $\pi_i$  are projections. For obvious reasons, we will use a sugared syntax when the desugaring is clear.

**Remark 2.2** Since expressions in this language are active, typical properties such as commutativity of addition are lost. It is possible to add a type **int** of passive (side-effect-free) expressions, and our semantic approach can handle these quite well [22]. But we have not obtained definability and full abstraction results in the presence of passivity. Among other things, the old problems with sequential functions [4] reappear.

**Remark 2.3** Active expressions are not necessary to raise the problem of irreversibility. For example, we can just use the command type, as in the block

```

begin
  integer x;
  x := 0;
  P(x := x + 1);
  if (x > 0) then diverge
end

```

with  $P : \mathbf{comm} \rightarrow \mathbf{comm}$ . This block is equivalent, in our language, to  $P(\mathbf{diverge})$ , i.e., has the same termination/nontermination behavior in all contexts. In a language with I/O or jumps these terms would be inequivalent. Then irreversibility would be exemplified not by a pure equivalence but as a more complex property (such as equivalence of termination behaviour, under the precondition that  $P$  does not perform a jump).

### 3 A Category of Object Spaces

In this section we will define the category of possible worlds based on the (free) object spaces of [22].

**Definition 3.1** Let  $A = (|A|, \circ_A)$  be a coherent space, i.e. a reflexive and symmetric binary relation  $\circ_A$  on a (countable) set  $|A|$ . The (free) object space  $\dagger A$  associated with  $A$  is the coherent space where  $|\dagger A| = |A|^*$  is the set of finite sequences of tokens in  $|A|$ , and  $a_1, \dots, a_n \circ_{\dagger A} b_1, \dots, b_m$  iff

$$\forall i \in \{1, \dots, \min(n, m)\}. a_1, \dots, a_{i-1} = b_1, \dots, b_{i-1} \implies a_i \circ b_i$$

The intuition in this definition is that tokens in  $\dagger A$  are “sequentialized.” One may think of a sequence  $a_1, \dots, a_n$  as representing a series of observations made on an object. The coherence relation  $\circ_{\dagger A}$  indicates when it is consistent to regard two traces as arising from the same computational object; see Example 3.5 below. Further motivation for the definition, based on a discussion relating to objects and automata, may be found in the appendix. A fuller treatment is in [22].

$X, Y, W$  will be used to range over the free spaces  $\dagger A$ . We will often consider  $X = |\dagger A|$  as a monoid, with unit (empty sequence)  $\epsilon_X$  and multiplication (concatenation) written simply by juxtaposition  $x_1 x_2$ .  $x_1 \cdots x_n$  will typically denote a multiplication where each  $x_i$  is a sequence, while  $a_1, \dots, a_n$  denotes a sequence of tokens  $a_i$ . We write singleton sequences as  $\langle a \rangle$  when necessary for disambiguation.

A regular map  $f : X \rightarrow Y$  of object spaces constructs a  $Y$ -object from an  $X$ -object by simulating the operations of the  $Y$ -object on the given  $X$ -object. Generally,  $f$  will be given by a relation  $f \subseteq |X| \times |Y|$  with elements written as  $x \mapsto y$ . Note that  $x$  and  $y$  are themselves sequences here. A pair  $x \mapsto y$  signifies that the  $Y$ -operation  $y$  is simulated by carrying out the operation  $x$  on an  $X$ -object. Now, we think of  $x \asymp y (= \neg(x \circ y) \vee x = y)$  as indicating that  $x$  and  $y$  possess the same “input information,” and we require that the input part of  $y$  determine the input part of  $x$ , i.e.,

$$y \asymp y' \implies x \asymp x'$$

Secondly, the output part of  $x$ , together with the input part of  $y$ , must determine the output part of  $y$ , i.e.,

$$x \circ x' \implies y \circ y'$$

These are standard conditions for linear functions. To these we add conditions concerning the preservation of monoid structure:

**Definition 3.2** A regular map  $f : X \rightarrow Y$  is a relation  $f \subseteq |X| \times |Y|$  such that, for all  $x_1 \mapsto y_1, x_2 \mapsto y_2 \in f$ ,

- (i)  $x_1 \circ x_2 \implies y_1 \circ y_2$ , and
- (ii)  $y_1 \asymp y_2 \implies x_1 \asymp x_2$ ,

satisfying

- (iii)  $\epsilon_X \mapsto \epsilon_Y \in f$ ,
- (iv)  $x_1 \mapsto y_1, x_2 \mapsto y_2 \in f \implies x_1 x_2 \mapsto y_1 y_2 \in f$ , and
- (v)  $x \mapsto y_1 y_2 \implies \exists x_1, x_2. x = x_1 x_2 \wedge x_1 \mapsto y_1, x_2 \mapsto y_2 \in f$ .

The condition (ii) can also be written as  $x_1 \circ x_2 \wedge y_1 = y_2 \implies x_1 = x_2$ . The conditions (iii-v) in the definition state that regular maps are state-independent or history-free. For example, the condition (iv) means that, if  $x_2 \mapsto y_2 \in f$ , signifying that an action  $y_2$  is simulated by  $x_2$ , then this simulation can always be tacked on “later,” on top of another simulation.

Even though our programming language is imperative, a form of history-freeness is appropriate in global maps because these correspond to denotations of closed terms. In a language obeying the stack discipline, state is securely encapsulated in local declarations  $\mathbf{new}(\lambda x.C)$ , so the closed terms themselves are effectively stateless. This viewpoint on global maps is also found in the possible world models [16,15].

**Definition 3.3** The category  $\mathbf{Ob}$  of (free) object spaces has as objects the spaces  $\dagger A$ . The morphisms are regular maps, with relational composition.

We can order the hom-sets of this category using the inclusion order of relations; this order corresponds to the stable order [2].

Commands are modelled using the space  $\dagger \mathbf{1}$ , where where  $\mathbf{1}$  is the one-token coherent space. The idea is that a command corresponds to an object with one operation, which when invoked simply runs the command. We write *comm* for  $\dagger \mathbf{1}$ .

Active integers are modelled using  $\dagger int$ , where  $int$  is the discrete coherent space of (non-negative) integers. Since any two integer tokens are inconsistent, all the tokens have the same “input part”. So, We write  $aint$  for  $\dagger int$ .

The opposite of  $int$  plays an “input” role in this category. The coherent space  $int^\perp$  has the same tokens as  $int$ , but all the tokens are considered consistent. We regard the information of a token as purely input. Intuitively, an object for  $\dagger int^\perp$  accepts an integer and uses it to potentially alter its internal state. We write  $acc$  for  $\dagger int^\perp$ .

**Remark 3.4** Let  $x \preceq y$  denote the relation  $\exists z. xz = y$ . Since  $y = ye$ , it follows that  $x \circ y$  whenever  $x \preceq y$ . In particular  $e \circ x$  for all  $x$ . An *object behavior* is a subset  $L \subseteq |X|$  that is left-closed (with respect to  $\preceq$ ) and pairwise-consistent. For example, the object behavior for **gensym** is the set of initial sequences  $1, \dots, n \in |aint|$ . By the results of Winskel [33], object behaviors form a dI-domain under the inclusion ordering. A regular map  $f : X \rightarrow Y$  determines a function  $\bar{f}$  from object behaviors of  $X$  to object behaviors of  $Y$ :

$$\bar{f}(L) = \{ y \in |Y| : \exists x \in L. x \mapsto y \in f \}$$

Such a function is *stable* (continuous and preserves consistent glb's) and *linear* (preserves all the lubs that exist).

There is an evident forgetful functor  $U : \mathbf{Ob} \rightarrow \mathbf{CohL}$  to the category of coherent spaces and linear maps. (Forget the conditions (iii-v) of regular maps.) This has a right adjoint whose object part is  $\dagger A$ :

$$\mathbf{CohL}(UX, A) \cong \mathbf{Ob}(X, \dagger A) \tag{1}$$

The morphism part of  $\dagger$  is given by

$$\dagger f = \{ a_1, \dots, a_n \mapsto b_1, \dots, b_n : a_i \mapsto b_i \in f, 1 \leq i \leq n \}$$

for  $f : A \multimap B$ . The adjunction gives a comonad  $U\dagger$  on coherent spaces (which we write as  $\dagger_L$  or simply  $\dagger$ ).

The category **Ob** has finite products. Recall first the definition of categorical product in the category **CohL** of coherent spaces and linear maps.

$$A \& B = (|A| + |B|, \circ_{A \& B}) \text{ with}$$

$$1.a \circ_{A \& B} 1.a' \iff a \circ_A a'$$

$$2.b \circ_{A \& B} 2.b' \iff b \circ_B b'$$

$$1.a \circ_{A \& B} 2.b \quad \text{always}$$

The product  $\dagger A \times \dagger B$  of object spaces is  $\dagger(A \& B)$ . (This is immediate from the fact that  $\dagger$  is a right adjoint.) The projections are  $\dagger \pi_i$ , for the projections  $\pi_i$  in **CohL**. For pairing, if  $f_i : \dagger C \rightarrow \dagger A_i$  then using adjunction  $U \dashv \dagger$  we obtain linear maps  $f'_i : \dagger C \rightarrow A_i$ ,  $i = 1, 2$ , and we can form their pair  $\langle f_1, f_2 \rangle : \dagger C \multimap A_1 \& A_2$  using product structure in **CohL**. Then using the adjunction again, we obtain a map  $\dagger C \rightarrow \dagger(A_1 \& A_2)$  which interprets pairing. The terminal object in **Ob** is  $I = \dagger emp$  where  $emp$  is the empty coherent space.

To model storage variables we use  $var = acc \times aint \cong \dagger(int^\perp \& int)$ . Intuitively, an object for this space has an operation of type  $acc$  for setting its value and an operation of type  $aint$  for reading the value. We regard the tokens of  $var$  as strings over  $\{put.i : i \in |int|\} \cup \{get.i : i \in |int^\perp|\}$  for mnemonic value.

The object behavior (cf., Remark 3.4)  $cell \subseteq |var|$  consists of those sequences  $t$  satisfying

$$t = (\dots get.i, get.i' \dots) \implies i = i'$$

$$t = (\dots put.i, get.i' \dots) \implies i = i'$$

$$t = (get.i \dots) \implies i = 0$$

This object behavior models a declared storage variable with initial value 0.

**Example 3.5** The coherence relation  $\bigcirc_{\dagger A}$  is meant to indicate *consistency* of observed behaviors. To illustrate this consider the case  $A = var$ , where we regard  $put.i$  tokens as input and  $get.i$  tokens as output. Two sequences  $a_1, \dots, a_n, a_{n+1}, a_{n+2} \dots$  and  $a_1, \dots, a_n, a'_{n+1}, a'_{n+2} \dots$  are coherent iff  $a_{n+1}$  and  $a'_{n+1}$  are coherent. The interesting case is when  $a_{n+1} \neq a'_{n+1}$ . If these are output tokens  $get.i$  and  $get.j$  then the sequences are incoherent, because  $i$  and  $j$  indicate different or inconsistent output observations (notice the implicit determinacy assumption). For coherence, if  $a_{n+1} \neq a'_{n+1}$  then one must be a  $put.j$  token. There is no inconsistency between an input action  $put.j$  and any other action because we do not (immediately) observe the (internal) result of the input action. Notice that there is no relationship between  $a_{n+2} \dots$  and  $a'_{n+2} \dots$ .

For example, consider two sequences  $put.2, get.7$  and  $put.4, get.9$ . The sequences differ coherently in the first position, and so are deemed coherent, even though they are incoherent in the second position. This is reasonable because we could certainly conceive of the following object: when given a  $put.2$  it changes its state to 7, when given a  $put.4$  it changes its state to 9, and when a  $get$  request is issued it simply returns the value of its internal state. So it is logically consistent to regard the two sequences as arising from the same object. This is why  $\bigcirc_{\dagger A}$  is defined so that sequences must be coherent only at the *first* place they differ. With different changes of state, such as in  $put.2$  and  $put.4$ , there is no inconsistency in having completely unrelated subsequent observations.

**Example 3.6** Some examples of regular maps are given in Table 1. The notation  $B_i(a)$  in the definition of  $cond_X$  means  $1.a$  if  $i = 0$  and  $2.a$  otherwise. Each of these maps may be understood as a simulation of the operations of one type on objects of another type. For example, the map  $seq$  simulates the unique operation of a command object ( $comm$ ) on an object with two command operations.

By virtue of isomorphism (1), many of the maps in Example 3.6 are uniquely determined by linear maps of coherent spaces: for instance,

$$seq_0 : \dagger(\mathbf{1} \& \mathbf{1}) \multimap \mathbf{1} = \{1.* , 2.* \mapsto *\}$$

---

$zero : I \rightarrow aint$	$= \{ \epsilon_I \mapsto \langle 0 \rangle^k : k \geq 0 \}$
$succ : aint \rightarrow aint$	$= \{ i_1, \dots, i_n \mapsto (i_1 + 1), \dots, (i_n + 1) : n \geq 0 \}$
$pred : aint \rightarrow aint$	$= \{ i_1, \dots, i_n \mapsto (i_1 - 1), \dots, (i_n - 1) : i_k > 0, 1 \leq j \leq n \}$
$skip : I \rightarrow comm$	$= \{ \epsilon_I \mapsto \langle * \rangle^k : k \geq 0 \}$
$seq : comm \times comm \rightarrow comm$	$= \{ (1.* , 2.*)^k \mapsto \langle * \rangle^k : k \geq 0 \}$
$deref : var \rightarrow aint$	$= \{ (get.i_1), \dots, (get.i_k) \mapsto i_1, \dots, i_k : k \geq 0 \}$
$assign : var \times aint \rightarrow comm$	$= \{ 2.i_1, 1.put.i_1, \dots, 2.i_k, 1.put.i_k \mapsto \langle * \rangle^k : k \geq 0 \}$
$cond_X : aint \times (X \times X) \rightarrow X$	$= \{ 1.i_1, 2.B_{i_1}(a_1) \dots 1.i_k, 2.B_{i_k}(a_k) \mapsto a_1, \dots, a_k : k \geq 0, i_n \in  int , a_n \in  A , X = \dagger A \}$

---

**Table 1** Examples of regular maps

We sum up some of this discussion, for the record.

**Lemma 3.7** *The category  $\mathbf{Ob}$  has finite products. The forgetful functor  $U : \mathbf{Ob} \rightarrow \mathbf{CohL}$  has a right adjoint  $\dagger$ .*

**Remark 3.8** The induced comonad  $\dagger_L$  on  $\mathbf{CohL}$  does *not* satisfy the isomorphism  $\dagger_L A \otimes \dagger_L B \cong \dagger_L(A \& B)$  characteristic of “!” in linear logic. The reason is that  $\dagger_L$  interleaves tokens from  $A \& B$ , and the order of interleaving is important.

**Remark 3.9** The category  $\mathbf{Ob}$  is the category of free coalgebras for  $\dagger_L$ , which is equivalent to the Kleisli category of  $\dagger_L$ . The definition of object spaces in [22] is more general, because it uses coalgebras other than the free ones. This is needed for closure under tensor products and for the treatment of passivity. But for the example programming language considered here the free coalgebras suffice.

Finally, we note an important property of the space  $aint$  of active integers: it is a generator for the category  $\mathbf{Ob}$ , in the following ordered sense.

**Lemma 3.10** *For maps  $f, g : X \rightarrow Y$  in  $\mathbf{Ob}$ ,*

$$f \sqsubseteq g \iff \forall e : aint \rightarrow X . e; f \sqsubseteq e; g .$$

**Proof.** The  $\implies$  direction is trivial. Conversely, suppose  $x \mapsto y$  is a pair in  $f$  that is not in  $g$ , where  $x = a_1, \dots, a_n$ . We want to find a map  $e : aint \rightarrow X$  such that  $e; f \not\sqsubseteq e; g$ . Treat  $a$  as a function  $\{1, \dots, n\} \rightarrow |X|$ . If  $\vec{v} = i_1, \dots, i_k \in \{1 \dots n\}^*$  is a string, write  $a(\vec{v})$  for  $a_{i_1}, \dots, a_{i_k}$ . Let  $e : \dagger int \rightarrow X$  be the regular

map  $\{\vec{i} \mapsto a(\vec{i}) : \vec{i} \in \{1 \dots n\}^*\}$ . To see that this is indeed a regular map, note that two strings  $\vec{i}$  and  $\vec{j}$  are consistent in  $\dagger int$  iff one of them (say  $\vec{i}$ ) is a prefix of the other. In that case  $a(\vec{i}) \preceq a(\vec{j})$  and we have  $a(\vec{i}) \subset a(\vec{j})$ . If, in addition,  $a(\vec{i}) = a(\vec{j})$  then  $\vec{i}$  and  $\vec{j}$  must be permutations of each other. Since  $\vec{i}$  is a prefix of  $\vec{j}$ , this means  $\vec{i} = \vec{j}$ . The other conditions of regular maps can be verified easily. Now,  $1, \dots, n \mapsto y$  is a pair in  $e; f$ , but not in  $e; g$ .  $\square$

This property will play a key role in connecting the model to the programming language, allowing the type **aint** to be used to generate distinguishing contexts.

## 4 Interference via Yoneda

The category **Ob** has a categorical product for modelling  $\times$  in our programming language. But it does not have an exponent, with  $\Lambda : \mathbf{Ob}(X \times Y, Z) \cong \mathbf{Ob}(X, Y \Rightarrow Z)$ . Intuitively, the problem is that a regular map  $f : X \times Y \rightarrow Z$  is a simulation using an  $X \times Y$ -object, i.e., an object with  $X$ - and  $Y$ -operations on some shared state. The currying transformation  $\Lambda$  would require us to separate the  $X$  and  $Y$  parts of the  $X \times Y$ -object. But they are not separable as they act on shared state.

To obtain the required interpretation, we embed this semantics (together with its treatment of first-order maps in Table 1) into a Cartesian closed category of functors using a Yoneda embedding. Thus, we interpret **comm** as  $\mathbf{Ob}(-, comm)$  and **aint** as  $\mathbf{Ob}(-, aint)$ , and the function type using the functor category exponent. The computational intuition underlying this reinterpretation is the following: We now regard an Algol command as a regular map  $W \rightarrow comm$ , where the role of  $W$  is something like that of the store parameter in traditional denotational semantics. A map  $W \rightarrow comm$  is the simulation of a command in a  $W$ -typed “store.” All Algol types are similarly parameterized by  $W$ 's, and this allows interference, or sharing, to be accounted for by considering meanings dependent on the *same* parameter  $W$ .

### 4.1 Domains, Functors, and the Yoneda Embedding

We will be working with an enriched version of the Yoneda embedding; see [8] for enriched notions. We use **Cpo** to denote the category of  $\omega$ -complete pointed posets and continuous functions, and **Cpo<sub>⊥</sub>** for the subcategory of strict functions. We refer to the objects simply as cpos.

Suppose **C** is a (small) **Cpo**-enriched category. This means that each hom set  $\mathbf{C}(X, Y)$  comes equipped with a cpo structure, and that composition is continuous with respect to this structure. **Cpo** itself has the obvious enriched structure. We can then look at enriched functors  $\mathbf{C}^{op} \rightarrow \mathbf{Cpo}$ , where  $\mathbf{C}^{op}$  uses the same ordering as **C**. In this case, enriched functors are simply ordinary functors whose action on the hom sets  $\mathbf{C}^{op}(X, Y) \rightarrow \mathbf{Cpo}(FX, FY)$  is continuous.

For any **C**-object  $X$  we have a contravariant hom functor  $\mathbf{C}(-, X) : \mathbf{C}^{op} \rightarrow$

**Cpo**. Then we have the following standard facts.

**Lemma 4.1** (i)  $\mathbf{C}(-, X)$  is a full and faithful **Cpo**-enriched functor that preserves any existing products in **C**;

(ii)  $\text{Nat}(\mathbf{C}(-, X), F) \cong F(X)$ , for any **Cpo**-enriched  $F : \mathbf{C}^{\text{op}} \rightarrow \mathbf{Cpo}$ .

**Proof.** The argument for 1 is standard. For 2 we proceed as usual, with the addition that we use enrichment to show continuity of the map  $\lambda g . F(g)x : \mathbf{C}(Y, X) \rightarrow F(X)$  obtained from an element  $x \in F(X)$ .  $\square$

Here, the hom set  $\text{Nat}(\cdot)$  is ordered pointwise. We refer to the second property as the Yoneda lemma.

**Definition 4.2** Given a small **Cpo**-enriched category **C**, the category  $\mathcal{M}_{\mathbf{C}}$  is defined as follows:

- **OBJECTS.** **Cpo**-enriched functors  $F : \mathbf{C}^{\text{op}} \rightarrow \mathbf{Cpo}$  that factor through the inclusion functor  $\mathbf{Cpo}_{\perp} \rightarrow \mathbf{Cpo}$ .
- **MORPHISMS.** All natural transformations of such functors.

The factoring condition is from [16]. Notice that the functors  $\mathbf{C}(-, X)$  satisfy this condition.

We can interpret typed  $\lambda$ -calculus and recursion in this category.

**Lemma 4.3**  $\mathcal{M}_{\mathbf{C}}$  is Cartesian closed  $(\times, I, \Rightarrow)$ . It has a least fixed-point combinator  $Y_A : (A \Rightarrow A) \rightarrow A$  for each functor  $A$  in  $\mathcal{M}_{\mathbf{C}}$ .

Products in  $\mathcal{M}_{\mathbf{C}}$  are defined pointwise as is usual in functor categories. The exponent is defined with the help of the Yoneda lemma. On **C**-objects,

$$(F \Rightarrow G)X = \text{Nat}(\mathbf{C}(-, X) \times F, G), \text{ ordered pointwise,}$$

and on morphisms, when  $f : Y \rightarrow X$ ,

$$(F \Rightarrow G)(f) p[Y']((g : Y' \rightarrow Y), a) = p[Y'](g; f, a).$$

Fixed-points are given by defining  $Y_A[X]p$  to be the least fixed-point of

$$\lambda a . p[X](id_X, a) : A(X) \rightarrow A(X).$$

$Y$  satisfies typical uniformity criteria for fixed-points, such as dinaturality.

**Remark 4.4** The role of the functor  $\mathbf{C}(-, X)$  in  $(F \Rightarrow G)X$  is just as in standard functor-category semantics, except that its order structure is also taken into account. This will allow certain of these hom functors to play a double role, used for quantification over contexts and for interpreting base types in the programming language. See lemma 6.2 for where this is used.

**Remark 4.5** Oles used the strictness condition on functors in order to obtain Cartesian closure. With hindsight, we can see this condition arising in another way. The category  $(\mathbf{Cpo}_{\perp})^{\text{Cpo}}$  is symmetric monoidal closed and there is an endofunctor  $! : (\mathbf{Cpo}_{\perp})^{\text{Cpo}} \rightarrow (\mathbf{Cpo}_{\perp})^{\text{Cpo}}$ , obtained by composing on the right with lifting, that has a comonad structure.  $\mathcal{M}_{\mathbf{C}}$  is equivalent to the Kleisli category of  $!$ . Thus, Oles's strictness condition arises naturally if we take  $\mathbf{Cpo}_{\perp}$  together with the lifting comonad as fundamental, and look for a model of intuitionistic linear logic based on functors into  $\mathbf{Cpo}_{\perp}$  rather than

looking directly for a model of intuitionistic logic (*cf.*, [20]).

#### 4.2 Semantic Model

The semantics is given in  $\mathcal{M}_{\mathbf{Ob}}$ . For the types, define

$$\begin{aligned} \llbracket \mathbf{comm} \rrbracket &= \mathbf{Ob}(-, \mathit{comm}) \\ \llbracket \mathbf{aint} \rrbracket &= \mathbf{Ob}(-, \mathit{aint}) \\ \llbracket s \times t \rrbracket &= \llbracket s \rrbracket \times \llbracket t \rrbracket \\ \llbracket s \rightarrow t \rrbracket &= \llbracket s \rrbracket \Rightarrow \llbracket t \rrbracket \end{aligned}$$

The defined type **var** gets the interpretation

$$\llbracket \mathbf{var} \rrbracket = \llbracket \mathbf{aint} \rightarrow \mathbf{comm} \rrbracket \times \llbracket \mathbf{aint} \rrbracket$$

Variables of this kind, ‘‘Algol variables,’’ can be more complicated than variable-objects-in-store-contexts, but note that the latter can be easily turned into Algol variables. Specifically, there is a natural injection  $\xi : \mathbf{Ob}(-, \mathit{var}) \rightarrow \llbracket \mathbf{var} \rrbracket$  defined by  $\xi[X](v) = (a, r)$  where

$$\begin{aligned} a[Y](f, e) &= \langle f; v, e \rangle; \mathit{assign} \\ r &= v; \mathit{deref} \end{aligned}$$

The data *assign* and *deref* may be found in Table 1.

The interpretations of first-order constants are obtained from the maps in Table 1 by the Yoneda embedding. For instance, **ifz** is interpreted by the composite map  $(\mathit{iso}; \mathbf{Ob}(-, \mathit{cond}_b))$ , where *iso* is the appropriate isomorphism

$$\mathbf{Ob}(-, \mathit{aint}) \times \mathbf{Ob}(-, b) \times \mathbf{Ob}(-, b) \xrightarrow{\mathit{iso}} \mathbf{Ob}(-, \mathit{aint} \times b \times b)$$

All that is left is to interpret **new** and **letval**.

To interpret **letval**, we define a map  $\mathit{letval} : \llbracket b \times (b \rightarrow b') \rrbracket \rightarrow \llbracket b' \rrbracket$ , which is determined uniquely by the following property:  $(x, \langle a \rangle) \in \mathit{letval}[X](p, q)$  iff

$$\begin{aligned} \exists x_1, x_2 \in |X|. \quad x_1 x_2 = x \wedge \\ \exists n \in |b|. \quad x_1 \mapsto \langle n \rangle \in p \wedge x_2 \mapsto \langle a \rangle \in q[X](\mathit{id}_X, k_n) \end{aligned}$$

(By focusing on a single output token  $\langle a \rangle$  we are essentially using the Kleisli representation of regular maps.) The idea is that we evaluate the argument  $p$ , consuming  $x_1$  from the state-context, and then we consume  $x_2$  while producing  $a$ .  $k_n \in \mathbf{Ob}(X, \mathit{aint})$  is the evident constantly- $n$  active integer (the unique map containing  $e_X \mapsto n$ ) in the case that  $b = \mathbf{aint}$ , and it is the constantly  $*$  command (**skip**) if  $b = \mathbf{comm}$ . Sending  $k_n$  as an argument to  $q$  shows how further evaluations of this argument always yield the same integer or command action.

For the semantics of **new** we need a map

$$\mathit{new} : (\llbracket \mathbf{var} \rrbracket \Rightarrow \llbracket \mathbf{comm} \rrbracket) \rightarrow \llbracket \mathbf{comm} \rrbracket$$

For every procedure  $p \in (\llbracket \mathbf{var} \rrbracket \Rightarrow \llbracket \mathbf{comm} \rrbracket)X$ ,  $\mathit{new}[X]p$  must be a regular map  $X \rightarrow \mathit{comm}$ . There are two main parts to obtaining such a map.  $\mathit{new}[X]$

must “call”  $p$  in an enlarged store type  $X'$  where there is an additional variable  $v \in \llbracket \mathbf{var} \rrbracket X'$ . This gives a command  $p^* : X' \rightarrow \mathit{comm}$ . Second,  $\mathit{new}[X]$  must convert  $p^*$  to a command  $X \rightarrow \mathit{comm}$  by supplying it with an appropriate enlarged store of type  $X'$ .

The first part is done by taking the space  $X' = X \times \mathit{var}$  and calling  $p$  with the variable obtained from the second projection  $\pi_2 : X \times \mathit{var} \rightarrow \mathit{var}$ :

$$p^* = p[X \times \mathit{var}](\pi_1, \xi[X \times \mathit{var}](\pi_2)) : X \times \mathit{var} \rightarrow \mathit{comm}$$

where  $\xi : \mathbf{Ob}(-, \mathit{var}) \rightarrow \llbracket \mathbf{var} \rrbracket$  is the embedding defined earlier. For the second part of converting  $p^*$  to  $X \rightarrow \mathit{comm}$ , define  $\mathit{new}[X]p$  as the unique regular map including the following pairs:

$$\{ x_0 \cdots x_k \mapsto \langle * \rangle : \exists s_1 \cdots s_k \in \mathit{cell}. x_0 s_1 x_1 \cdots s_k x_k \mapsto \langle * \rangle \in p^* \}$$

Again, this is the Kleisli representation which, by the adjunction  $U \dashv \dagger$ , determines the map completely. We are using the monoid multiplication (juxtaposition) here, so, for example, some of the sequences  $x_i$  may well be empty. The idea of this definition is that the uses  $s_i$  of the local variable are simply ignored at the non-local level. Note that while we can convert commands  $X \times \mathit{var} \rightarrow \mathit{comm}$  to  $X \rightarrow \mathit{comm}$ , we do not have a corresponding regular map  $X \rightarrow X \times \mathit{var}$ . Indeed, since regular maps are history-free they cannot create new objects.

This completes the definition of the model.

**Example 4.6** Consider the application map  $\mathbf{app} : \llbracket (\mathbf{aint} \rightarrow \mathbf{aint}) \times \mathbf{aint} \rrbracket \rightarrow \llbracket \mathbf{aint} \rrbracket$ . On the level of functor categories the definition is  $\mathbf{app}[X](p, a) = p[X](\mathit{id}_X, a)$ . On the level of object-spaces, the effect is as follows. Applying the Yoneda lemma a number of times, we find that this application map determines a continuous function

$$\begin{aligned} \llbracket (\mathbf{aint} \rightarrow \mathbf{aint}) \times \mathbf{aint} \rrbracket X &\longrightarrow \llbracket \mathbf{aint} \rrbracket X \\ &\cong \mathbf{Ob}(X \times \mathit{aint}, \mathit{aint}) \times \mathbf{Ob}(X, \mathit{aint}) \longrightarrow \mathbf{Ob}(X, \mathit{aint}) \end{aligned}$$

The induced function takes a pair  $(p, f)$  of maps and produces a regular map  $X \rightarrow \mathit{aint}$ .

$$X \xrightarrow{\langle \mathit{id}, f \rangle} X \times \mathit{aint} \xrightarrow{p} \mathit{aint}$$

This composite is the unique regular map containing the following pairs:  $x_0 y_1 x_1 \cdots y_n x_n \mapsto \langle a \rangle$  whenever there is  $k_1 \cdots k_n \in |\mathit{aint}|$  such that

$$y_i \mapsto \langle k_i \rangle \in f \quad \wedge \quad x_0 k_1 x_1 \cdots k_n x_n \mapsto \langle a \rangle \in p.$$

This is the form of sharing or interference that we obtain by “placing objects into the same context,” the common context here being  $X$ . The  $x_i$  and  $y_j$  in  $x_0 y_1 x_1 \cdots y_n x_n$  represent interleaved uses of  $X$  by  $p$  and  $f$ . Thus, the Yoneda embedding leads not only to a treatment of function types that is technically correct, but an implementation of sharing that is intuitively reasonable (and which has proven difficult to come by otherwise).

**Remark 4.7** It is perhaps surprising that a category  $\mathbf{C}^{op}$  can be used as the category of worlds, where  $\mathbf{C}$  is a category of functions. In previous work [24,16,30], the categories of worlds typically involved morphisms that were

more than (even opposites of) functions; they were pairs of functions, one for de-allocation of storage variables, and one for overwriting “small” pieces of “large” states. In an explicit-state setup, when modelling commands as state-to-state functions, both the co- and contravariant roles of state need to be accounted for in order to get a functor of command meanings. The completely contravariant account given here via  $\mathbf{C}^{op}$ , using only de-allocations (Weakenings) to interpret declarations, is possible because of the “demand-driven” nature of the treatment of commands in [22].

## 5 First-order definability

We know that the spaces  $\text{Nat}(\llbracket s \rrbracket, \llbracket t \rrbracket)$  of natural transformations are cpos, but to study definability in the model we need more information on their structure. In this section we use the Yoneda lemma to calculate the structure precisely, by showing that for base types  $s$  and  $t$  these cpos are algebraic. In fact, we show much more: each of these cpos is isomorphic to (the set of points of) a coherent space.

Given this characterization, we move on to show that every finite element in these domains is definable by a closed term in the programming language. By algebraicity, every element is then the lub of definable ones. Standard CCC manipulations allow us to obtain an analogous result for all global elements  $I \rightarrow \llbracket t \rrbracket$ , where  $t$  is an arbitrary first-order type. (The *order* of a type is defined inductively:  $\text{order}(\mathbf{aint}) = \text{order}(\mathbf{comm}) = 0$ ,  $\text{order}(s \times t) = \max(\text{order}(s), \text{order}(t))$  and  $\text{order}(s \rightarrow t) = \max(\text{order}(s) + 1, \text{order}(t))$ .)

**Lemma 5.1** *Suppose  $b_1, \dots, b_n, b$  are base types. Then*

$$\text{Nat}(\llbracket b_1 \times \dots \times b_n \rrbracket, \llbracket b \rrbracket)$$

*with pointwise order is isomorphic to a coherent space.*

In the statement of the lemma, and throughout, we confuse a coherent space with the cpo of its points, ordered by inclusion [5].

**Proof.** Let  $A_i$  and  $B$  be the coherence spaces used in the interpretations of  $b_i$  and  $b$ , *int* in the case of  $\mathbf{aint}$ ,  $\mathbf{1}$  in the case of  $\mathbf{comm}$ . Let us calculate.

$$\begin{aligned} & \text{Nat}(\llbracket b_1 \times \dots \times b_n \rrbracket, \llbracket b \rrbracket) \\ & \cong \text{Nat}(\mathbf{Ob}(-, \dagger A_1) \times \dots \times \mathbf{Ob}(-, \dagger A_n), \mathbf{Ob}(-, \dagger B)) && \text{definition} \\ & \cong \text{Nat}(\mathbf{Ob}(-, \dagger A_1 \times \dots \times \dagger A_n), \mathbf{Ob}(-, \dagger B)) && \text{Yoneda preserves } \times \\ & \cong \mathbf{Ob}(\dagger A_1 \times \dots \times \dagger A_n, \dagger B) && \text{Yoneda lemma} \\ & \cong \mathbf{Ob}(\dagger(A_1 \& \dots \& A_n), \dagger B) && \text{definition} \\ & \cong \mathbf{CohL}(\dagger_L(A_1 \& \dots \& A_n), B) && U \dashv \dagger, \dagger_L = U \dagger \end{aligned}$$

□

**Proposition 5.2** *Given base types  $b_1, \dots, b_n$  and  $b$ , any finite element in  $\text{Nat}(\llbracket b_1 \times \dots \times b_n \rrbracket, \llbracket b \rrbracket)$  is definable by a term-in-context*

$$x_1 : b_1, \dots, x_n : b_n \vdash Q : b$$

**Proof.** We use the representation in terms of  $\dagger(A_1 \& \cdots \& A_n) \multimap B$ , and consider tokens of  $A_1 \& \cdots \& A_n$  as of the form  $i.a$  for  $1 \leq i \leq n$ , the  $i$  indicating the component. Let  $f$  be a finite linear map. Define the size of  $f$  to be the number of tokens (of  $A_1, \dots, A_n$  and  $B$ ) in its trace. The proof is by induction on the size of  $f$ . There are three cases.

1.  $f = \emptyset$ . Then  $Q = \Omega$ , some divergent term of type  $b$ .
2.  $\epsilon \mapsto a \in f$ . Coherence of  $f$  implies that  $f = \{\epsilon \mapsto a\}$ . If  $b$  is **comm**, let  $Q = \mathbf{skip}$ . If  $b$  is **aint**, then let  $Q = \mathbf{succ}^a(0)$ .
3.  $(i.a)s \mapsto b \in f$ . Coherence of  $f$  means that if  $(i'.a')s' \mapsto b' \in f$ , then  $i = i'$ . Suppose that  $b_i$  is **aint**. Let  $z$  be the collection of those  $a'$  where  $(i.a')s' \mapsto b' \in f$ , for some  $s', b'$ . Since  $f$  is finite,  $z$  is finite. For each  $a' \in z$ , let  $f_{a'} = \{s' \mapsto b' : (i.a')s' \mapsto b' \in f\}$ . By induction,  $f_{a'}$  is definable by a term  $M_{a'}$ . Let  $k_1, \dots, k_n$  be an enumeration of  $z$ . Note that  $z$  is not empty. Then  $f$  is definable by the following term, using evident notation for **if**, where  $x_i : \mathbf{aint}$  is the identifier corresponding to  $b_i$ .

```

letval  $x_i$  ( $\lambda m : \mathbf{aint}$ 
  if  $m = k_1$  then  $M_{k_1}$ 
  else if  $m = k_2$  then  $M_{k_2}$ 
   $\vdots$ 
  else if  $m = k_n$  then  $M_{k_n}$ 
  else  $\Omega$ 
)

```

If  $b_i$  is **comm** the proof is simpler. □

Notice that there is a form of sequentiality at work in case 3 of the proof. Coherence of a finite element  $f$  means that if  $(i', a')s' \mapsto b' \in f$  and  $(i, a)s \mapsto b \in f$ , then  $i = i'$ . This corresponds to the intuition that the  $i$ 'th component is queried first by  $f$ , which is why we are accounting properly for sequential facilities (at first order). The active nature of the arguments is crucial here, as this kind of account of sequentiality doesn't adapt to PCF-style computation.

**Corollary 5.3** *For any first-order type  $t$ ,  $\mathbf{Nat}(I, \llbracket t \rrbracket)$  is isomorphic to a coherent space, each of whose finite elements is definable by a closed term  $\vdash Q : t$ .*

**Proof.** From the proposition, using standard (syntactic versions of) CCC manipulations involving currying, pairing, and the cartesian isomorphism  $A \rightarrow (B \times C) \cong (A \rightarrow B) \times (A \rightarrow C)$ . □

**Example 5.4** Closed terms of type **comm**  $\rightarrow$  **comm** are interpreted as elements of  $\mathbf{Nat}(I, \llbracket \mathbf{comm} \rightarrow \mathbf{comm} \rrbracket)$ . Let us calculate this hom set using the

argument in the proof of lemma 5.1.

$$\begin{aligned}
& \text{Nat}(I, \llbracket \mathbf{comm} \rightarrow \mathbf{comm} \rrbracket) \\
& \cong \text{Nat}(\llbracket \mathbf{comm} \rrbracket, \llbracket \mathbf{comm} \rrbracket) && \text{(enriched) CCC isomorphism} \\
& \cong \text{Nat}(\mathbf{Ob}(-, \dagger 1), \mathbf{Ob}(-, \dagger B)) && \text{definition} \\
& \cong \mathbf{Ob}(\dagger 1, \dagger 1) && \text{Yoneda lemma} \\
& \cong \mathbf{CohL}(\dagger_L(1), 1) && U \dashv \dagger, \dagger_L = U \dagger \\
& \cong N_{\perp} && \text{calculation}
\end{aligned}$$

where  $N_{\perp}$  is the flat natural numbers (see [22]). Each  $n \in N$  corresponds to a Church numeral  $\lambda c.c^n$  of type  $\mathbf{comm} \rightarrow \mathbf{comm}$ , where  $c^0 = \mathbf{skip}$  and  $c^{i+1} = c; c^i$ . The least element this type is the divergent command  $\mathbf{Y}(\lambda c.c)$ . Thus, *every* element in the hom set  $\text{Nat}(I, \llbracket \mathbf{comm} \rightarrow \mathbf{comm} \rrbracket)$  is definable.

This representation of  $\text{Nat}(I, \llbracket \mathbf{comm} \rightarrow \mathbf{comm} \rrbracket)$  should be compared to [15,26], where the corresponding representation yields  $N_{\perp} \otimes Vnat^{op}$  with  $Vnat^{op}$  the upside-down vertical natural numbers and  $\otimes$  the smash product. The  $Vnat^{op}$  component has entirely to do with snapback operations which, in this case, lead to a more complex domain.

## 6 A Full Abstraction Result

In reasoning about second-order terms we need to consider the denotations of first-order types at various possible worlds, and not only global elements  $I \rightarrow \llbracket t \rrbracket$  for first-order  $t$ . Syntactically, this corresponds to the fact that the context lemma does not hold in our example language: one needs more than closed applicative contexts to distinguish closed terms of functional type. Semantically, it corresponds to the fact that the category is not well pointed: to distinguish parallel maps  $f, g : \llbracket s \rrbracket \rightarrow \llbracket t \rrbracket$  it is not enough to compose on the left with maps  $I \rightarrow \llbracket s \rrbracket$  out of the terminal object. So the definability result of the previous section does not immediately give us full abstraction for closed terms of second order.

To get full abstraction at second order, we first show that, for the appropriate types, different natural transformations can be distinguished at the possible world *aint*. This then enables us to use the programming language type **aint**, together with **new**, to build distinguishing contexts. It suffices to consider applicative contexts with a single free identifier of type **aint**, wrapped in the scope of a new variable declaration used to generate an active integer to bind to this free identifier.

**Lemma 6.1**  $\eta \sqsubseteq \mu : \llbracket s \rightarrow t \rrbracket \rightarrow \llbracket t' \rrbracket \iff \eta[\mathit{aint}] \sqsubseteq \mu[\mathit{aint}]$ , for 0-order types  $t, s, t'$ .

**Proof.** The  $\implies$  direction is trivial. Suppose  $\eta \not\sqsubseteq \mu$ . Then for some  $X$ ,  $\eta[X] \not\sqsubseteq \mu[X] : \mathbf{Ob}(X \times A_s, A_t) \rightarrow \mathbf{Ob}(X, A_{t'})$ , using a representation of the types calculated as in lemma 5.1. Consider  $f \in \mathbf{Ob}(X \times A_s, A_t)$  where  $\eta[X]f \not\sqsubseteq \mu[X]f$ . By Lemma 3.10 there exists a map  $e : \mathit{aint} \rightarrow X$  such that

$e; \eta[X]f \not\sqsubseteq e; \mu[X]f$ . Naturality of  $\eta$  and  $\mu$  with respect to  $e$  then implies that  $\eta[\mathbf{aint}]((e \times id); f) \not\sqsubseteq \mu[\mathbf{aint}]((e \times id); f)$  and we are done.  $\square$

Next, we want a definability result about first-order types instantiated at world  $\mathbf{aint}$ .

**Lemma 6.2** *For order 0 types  $s$  and  $t$ ,  $\llbracket s \rightarrow t \rrbracket X$  is isomorphic to a coherent space. Further, each finite element of  $\llbracket s \rightarrow t \rrbracket \mathbf{aint}$  is definable (in an evident sense) by a term-in-context  $y : \mathbf{aint} \vdash M : s \rightarrow t$ .*

The term  $M$  determines an element of  $\llbracket s \rightarrow t \rrbracket \mathbf{aint}$  using the isomorphism

$$\llbracket s \rightarrow t \rrbracket \mathbf{aint} \cong \text{Nat}(\llbracket \mathbf{aint} \rrbracket, \llbracket s \rightarrow t \rrbracket).$$

This is where  $\mathbf{Ob}(-, \mathbf{aint})$  is playing a double role, used in the definition of  $\llbracket s \rightarrow t \rrbracket \mathbf{aint}$  and as the interpretation of  $\mathbf{aint}$  in  $\text{Nat}(\llbracket \mathbf{aint} \rrbracket, \llbracket s \rightarrow t \rrbracket)$ .

**Proof.** We can calculate the domain explicitly using a Yoneda lemma argument again, as in lemma 5.1.

$$\begin{aligned} & \llbracket s \rightarrow t \rrbracket X \\ & \cong \text{Nat}(\mathbf{Ob}(-, X) \times \llbracket s \rrbracket, \llbracket t \rrbracket) \text{ definition} \\ & \cong \mathbf{Ob}(X \times \dagger A', \dagger A) \quad \text{as before} \\ & \cong \mathbf{CohL}(U(X \times \dagger A'), A) \quad \text{by (1)} \end{aligned}$$

This gives the first part of the lemma.

In the case that  $X = \mathbf{aint} = \dagger \mathbf{int}$  we use the definition of product in  $\mathbf{Ob}$  to obtain the representation

$$\mathbf{CohL}(\dagger_L(\mathbf{int} \& A'), A)$$

Once again,  $A'$  and  $A$  are the coherent spaces used in the interpretation of  $s$  and  $t$  (possibly applying product-preservation of  $\dagger$ ).

To define the finite elements of this domain, recall that we have seen that it is isomorphic as a cpo to the space of natural transformations  $\text{Nat}(\llbracket \mathbf{aint} \times s \rrbracket, \llbracket t \rrbracket)$  and we have already shown that these finite elements are definable by terms-in-context

$$y : \mathbf{aint}, x : s \vdash Q' : t$$

The desired term-in-context

$$y : \mathbf{aint} \vdash Q : s \rightarrow t$$

defines the corresponding finite element of  $\llbracket s \rightarrow t \rrbracket \mathbf{aint}$ .  $\square$

**Theorem 6.3 (Inequational Full Abstraction)**

*If  $\vdash M : (t_1 \times \cdots \times t_n \rightarrow t) \rightarrow t'$  and  $\vdash N : (t_1 \times \cdots \times t_n \rightarrow t) \rightarrow t'$  are closed terms of second-order type, then*

$$\llbracket M \rrbracket \sqsubseteq \llbracket N \rrbracket \iff \forall C[\cdot]. \llbracket C[M] \rrbracket \sqsubseteq \llbracket C[N] \rrbracket$$

Here  $C[\cdot]$  ranges over ground contexts.

**Proof.** Only the  $\Leftarrow$  direction needs to be proven. Suppose  $\llbracket M \rrbracket \not\sqsubseteq \llbracket N \rrbracket$ . We will construct a command-typed context  $C[\cdot]$  where  $\llbracket C[M] \rrbracket \not\sqsubseteq \llbracket C[N] \rrbracket$ .

Since  $M$  and  $N$  are closed terms, they determine natural transformations  $\llbracket t_1 \times \cdots \times t_n \rightarrow t \rrbracket \rightarrow \llbracket t' \rrbracket$ . Using Lemma 6.1, Lemma 6.2 (algebraicity of  $\llbracket t_1 \times \cdots \times t_n \rightarrow t \rrbracket \text{aint}$ ) and continuity we may calculate

$$\begin{aligned} & \llbracket M \rrbracket \not\sqsubseteq \llbracket N \rrbracket \\ \Rightarrow & \llbracket M \rrbracket[\text{aint}] \not\sqsubseteq \llbracket N \rrbracket[\text{aint}] \\ \Rightarrow & \exists \text{ finite } d \in \llbracket t_1 \times \cdots \times t_n \rightarrow t \rrbracket \text{aint}. \llbracket M \rrbracket[\text{aint}]d \not\sqsubseteq \llbracket N \rrbracket[\text{aint}]d \end{aligned}$$

By Lemma 6.2 there is  $y : \mathbf{aint} \vdash Q : t_1 \times \cdots \times t_n \rightarrow t$  that defines  $d$ .

Next, given such a  $d$ , we know that the trace sets of  $\llbracket M \rrbracket[\text{aint}]d$  and  $\llbracket N \rrbracket[\text{aint}]d$  in  $\llbracket \mathbf{comm} \rrbracket \text{aint} \cong \mathbf{CohL}(\dagger \text{int}, \mathbf{1})$  are such that  $(\llbracket M \rrbracket[\text{aint}]d) \not\sqsubseteq (\llbracket N \rrbracket[\text{aint}]d)$ , say  $s \mapsto * \in \llbracket M \rrbracket[\text{aint}]d$  and  $s \mapsto * \notin \llbracket N \rrbracket[\text{aint}]d$ . We construct a term-in-context  $x : \mathbf{var} \vdash c : \mathbf{aint}$  as follows. If  $s = k_1 \cdots k_n$  then  $c$  is the term

$$\begin{aligned} & x := x + 1; (\text{if } x = 1 \text{ then } k_1 \\ & \quad \text{else if } x = 2 \text{ then } k_2 \\ & \quad \quad \vdots \\ & \quad \text{else if } x = n \text{ then } k_n \\ & \quad \text{else } \Omega \\ & ) \end{aligned}$$

Here recall that we are using a sequencing combinator  $C;E$  as sugar for  $\mathbf{letval} C (\lambda z.E)$  where  $z$  is not free in  $C$  or  $E$ . If  $s = \epsilon$  then  $c$  is  $\Omega$  (it doesn't matter what  $c$  is in this case).

With this  $c$  and  $Q$ , a context distinguishing  $M$  and  $N$  is

$$\begin{aligned} & \mathbf{new} x.x := 0; \\ & (\lambda y : \mathbf{aint} . [\cdot] Q)c \end{aligned}$$

□

We have formulated the full abstraction result for second-order functions that take a single first-order function as an argument. It should be clear from the form of the proof that the argument works for all second-order types. We do not know what happens at higher types.

**Example 6.4** We illustrate the semantics for the example from the Introduction. First, we have a regular map  $\mathit{gensym} : \mathbf{var} \rightarrow \mathbf{aint}$  that builds the behavior of  $\mathbf{gensym}$  by simulating its output in terms of  $\mathbf{var}$ -typed actions.  $\mathit{gensym}$  is given by

$$\begin{aligned} & \{ \mathit{get}.i_1, \mathit{put}.(i_1 + 1), \mathit{get}.j_1, \dots, \mathit{get}.i_n, \mathit{put}.(i_n + 1), \mathit{get}.j_n \mapsto j_1, \dots, j_n : \\ & \quad i_k, j_k \in |\mathbf{int}| \} \end{aligned}$$

As mentioned in Remark 3.4, the regular map determines a function from object behaviors of  $\mathbf{var}$  to those of  $\mathbf{aint}$ . In particular, when applied to the behavior  $\mathit{cell} \subseteq |\mathbf{var}|$ , the function gives an object behavior  $\{ \langle 1, \dots, n \rangle : n \in |\mathbf{int}| \}$  of type  $\mathbf{aint}$ . This corresponds to how  $\mathbf{gensym}$  is defined in terms of a declared variable.

The meaning of the block

```

begin
  integer  $x$ ;
  integer procedure gensym; {  $x := x + 1$ ; return( $x$ ); }
   $x := 0$ ;
   $P(\mathbf{gensym})$ ;
  if ( $\mathbf{gensym} > 1$ ) then diverge
end

```

is a natural transformation of type  $(\llbracket \mathbf{aint} \rrbracket \Rightarrow \llbracket \mathbf{comm} \rrbracket) \rightarrow \llbracket \mathbf{comm} \rrbracket$ . Its action at a possible world  $W$  is a continuous function

$$\text{Nat}(\mathbf{Ob}(-, W) \times \llbracket \mathbf{aint} \rrbracket, \llbracket \mathbf{comm} \rrbracket) \rightarrow \mathbf{Ob}(W, \text{comm})$$

which, using a Yoneda lemma calculation, reduces to a continuous function  $f : \mathbf{Ob}(W \times \text{aint}, \text{comm}) \rightarrow \mathbf{Ob}(W, \text{comm})$ . The action of  $f$  on a regular map  $p : W \times \text{aint} \rightarrow \text{comm}$  may be calculated as the following map:

$$\begin{aligned}
& \{ w_1 \cdots w_n \mapsto * : \exists t \in \text{cell}, t' \in |\text{var}|, i \in |\text{int}|, s_1, \dots, s_n \in |\text{aint}|. \\
& \quad t = \langle \text{put}.0 \rangle t' \langle \text{get}.i, \text{put}.(i+1), \text{get}.(i+1) \rangle \wedge i+1 \leq 1 \wedge \\
& \quad (t' \mapsto s_1 \cdots s_n) \in \text{gensym} \wedge \\
& \quad (w_1 s_1 \cdots w_n s_n \mapsto *) \in p \}
\end{aligned}$$

(For clarity, we have shown a linear map of type  $W \multimap \mathbf{1}$ . The corresponding regular map  $W \rightarrow \text{comm}$  is obtained by “iterating” this behavior.) The sequence  $t$  denotes the operations performed on the variable  $x$ . Given that the final value of  $x$  must be no greater than 1, the sequences  $t'$  and  $s_1 \cdots s_n$  can only be empty. Thus, the linear map is equal to

$$\{ w \mapsto * : w \mapsto * \in p \}$$

It is clear that the meaning of  $P(\mathbf{diverge})$  maps  $p$  to precisely the same regular map.

We must admit that the reasoning in this example is rather technical. Nevertheless, it illustrates an interesting feature of the object-based semantics. After applying a Yoneda lemma argument, we see that the (denotation of) procedure  $P$  is a regular function  $W \times \text{aint} \rightarrow \text{comm}$ , with  $W$  corresponding to the context of evaluation and  $\text{aint}$  to the argument. The semantics in this case works by “communication” between the procedure  $P$  and the local block. Where  $P$  expects an argument of type  $\text{aint}$ , the block simulates the argument in terms of the  $\text{var}$ -typed behavior  $\text{cell}$ . The interesting point is that the domain  $W \times \text{aint}$  for  $P$  does not mention the space  $\text{var}$  corresponding to local variable  $x$  at all, or for that matter any other type that may be used in a simulation of the  $\text{aint}$  argument. This corresponds to the intuition that any meaning for procedure  $P$  is defined without reference to the local variable.

## 7 Related Work

Although there has been a good deal of theoretical work on the foundations of object-oriented programming, most of it has concentrated on typing issues in a purely-functional context (see, for example, [6]) and so bears little relation to our work. For us, the initial conception of object involves at least a hidden local state together with operations acting upon it.

Much closer to our concerns is work on translating objects into process calculi, e.g., [32,7]. In this approach an object is treated as a process of a certain form, with the state implicit in the history of events; this aspect is clearly related, in pre-theoretic conception, to the approach of [22]. But the results and details are difficult to compare. Here the focus has been on denotational methods, and examining the connection (full abstraction) with an example programming language. In comparison, the process approach can be thought of as being broader (handling more features) but, as far as we are aware, no analysis indicating the accuracy of the resultant encodings has yet been given.

Closer still to our concerns is a variety of applications and extensions of functor category semantics. One of these is the work of Pitts and Stark on dynamic allocation [19], where a language is considered in which mere equality of names is the basic operation besides local allocation; they obtain a full abstraction result for first-order types. (Equality of names or locations does not fit so easily into the object-based models, which follow Reynolds’s lead [24] in taking a location-free view of state.) It does not appear that the phenomenon of irreversibility arises in this very bare setting of local names, but neither is it certain that actual *storable* values are necessary for mild cases of irreversibility to arise. For instance, something similar appears to be present in a simple form in the language SPCF of [3], though we are unsure of the exact relationship.

Sieber has built a model for an Algol-like language in which functors are equipped with logical relations that are used to constrain function types [26], and has obtained a full abstraction result for the closed terms of second-order. The proof is subtle and original, making use of “finitely determined” natural transformations; it is not obvious whether the cpo’s in Sieber’s semantics are even algebraic. The proof given here is much less sophisticated, using the usual method of definability of finite elements.

There are important differences between our language and the one in [26]. First and foremost is that Sieber’s results are for a language with a snapback combinator: so, in comparison to the work reported here, we may say that his model accounts for locality to a good degree, but not for irreversibility. Another difference is that Sieber’s language has a form of side effect-free integer expression, whereas we have used active integers. Our model can easily be extended to deal with passive integers, but in that case we have not obtained a full abstraction result: the old problems with sequential functions crop up again [4].

But we should emphasize that, though it does not have passive integers,

the language considered here *is* sequential; it is one where the order of evaluation of (at least base-type) arguments can be recorded using storage variables (*cf.* [3]). It would seem to make sense to try to push this explanation of “active sequentiality,” utilizing coherent spaces and the stable order (on regular functions), as far as possible before abandoning coherent spaces. And of course full abstraction is not the ultimate aim of the semantics, though in the course of proving the result we did find legitimate structure associated with imperative types (this structure is of more interest than the result itself, which is only a technical indicator).

The parametricity models (based on PERs and logical relations) presented in [15] do not account for irreversibility, either. However, we may understand the main message of that work as applying more broadly than to the specific models. The proposal there was that the abstractness of local state could be understood in terms of Strachey’s concept of parametric (uniform) polymorphism [28,25]. This leads to quite a convincing explanation of locality. Furthermore, it has recently become clear that a slight variation on the parametricity semantics, based on a strict function model of linear (even, relevant) polymorphism, rules out the snapback and other unwanted operations. There should be close connections between the parametric and object-based semantics.

We expect that some readers will feel (with us) that the model here works in a slightly mysterious fashion, without providing an “explanation” of locality and irreversibility. The methods of building up computational entities in the model do not mention any conditions related to these properties. The properties (to the extent we know what they are) arise as a consequence of the way objects are constructed. It may simply be that an axiomatic approach to these issues, focusing more on *properties* characteristic of locality and irreversibility, is best carried out within the context of an explicit-state semantics, though this is by no means certain. In any event, we have shown that the model is quite accurate, and so we expect that such an “explanation” should also be consistent with the object-based semantics.

Ultimately, we do not believe that there should be a conflict between the explicit state view, as exemplified by the the parametricity models, and the view of state as implicit in histories of events. Very often, it is most efficient to conceive of objects as computational entities with pieces of state and operations, though at other times it can be more efficient to work directly in terms of traces or similar representations. For instance, here we were able to calculate the domain-theoretic structure of types with great ease, while the principles explicitly adopted in the parametricity models often (but not always) lead to smoother reasoning about specific examples. Ideally one would hope to have precise means of linking these two forms of description, enabling passage back and forth between one and the other. These connections await further development.

## Acknowledgement

We are grateful to Bob Tennent for comments and discussions.

## Appendix

### A Objects, Coherent Spaces, and Automata

We think of an object as a computational entity with a mutable internal store and a collection of observable operations that can read and alter the store. In this respect, objects are much like automata. Elementary notions of objects can receive some illumination by comparing to concepts of automata theory.

Let  $M = (M, \cdot, e_M)$  be a *monoid*, i.e., a set with an associative operation “ $\cdot$ ” and a unit  $e_M$  for this operation. (We often write a product  $x \cdot y$  as simply  $xy$ .) An *automaton* for  $M$  is a pair  $\alpha = (Q, \alpha : Q \times M \rightarrow Q)$  where  $Q$  is a set (of *states*) and  $\alpha$  is a partial function (the *transition function*) satisfying

$$\begin{aligned} \alpha(q, e_M) &= q \\ \alpha(q, xy) &= \alpha(\alpha(q, x), y) \end{aligned} \tag{A.1}$$

(These identities are understood in the context of partial functions: if either side is defined, the other side is defined and equal.) Such automata are also called “monoid actions” or “M-sets” (with partial functions).

Often, one takes  $M$  to be a free monoid  $\Sigma^*$ , the set of strings over an “alphabet”  $\Sigma$ . In this case,  $\alpha$  is uniquely determined by giving a “one-step” transition function  $\alpha_0 : Q \times \Sigma \rightarrow Q$ . If  $\Sigma$  is a one-element set  $\mathbf{1} = \{*\}$ , then  $\alpha_0$  reduces to a function  $Q \rightarrow Q$ . An automaton of this form can be regarded as a “command object,” an object with a single operation that alters the store. Another example is an “active integer” that returns 2 the first time it is used, 4 the second, and continues doubling its value thereafter. As an automaton, one representation is obtained by taking  $Q = \{1, 2, 3, \dots\}$ ,  $\Sigma = \{2, 4, 8, 16, \dots\}$  and setting then  $\alpha_0(n, 2^n) = n + 1$ . This active integer illustrates the irreversibility of state changes mentioned earlier: we never return to the state 1 after an initial use of the object (assuming 1 as the initial state). The representation of active integers as automata is not entirely satisfactory in that we would like to think of the integer as an “output” of the automaton, but nothing in the definition suggests this. This is remedied below in the definition of objects with reference to coherent spaces.

If  $\alpha : Q \times M \rightarrow Q$  is an automaton, its *behavior* at a state  $q \in Q$  is defined as

$$L_\alpha(q) = \{x \in M : \alpha(q, x) \text{ is defined}\}$$

$L_\alpha(q)$  is “left-closed,” i.e.,  $xy \in L_\alpha(q) \implies x \in L_\alpha(q)$ . By the second identity of (A.1), if  $\alpha(q, xy)$  is defined,  $\alpha(q, x)$  must be defined. Conversely, given any left-closed subset  $X \subseteq M$ , we can recover an automaton from it though not uniquely. A canonical choice is to take  $Q = X$  and define  $\alpha : X \times M \rightarrow X$  by

$$\alpha(x, y) = xy$$

Then,  $L_\alpha(e_M) = X$ . This is the initial automaton with behavior  $X$ . The final automaton is obtained by identifying all the right-congruent elements in  $X$ . These two automata sandwich all the other automata with behavior  $X$  (at designated start states).

This discussion illustrates how we might regard automata as *intensions* and their behaviors as *extensions*. We can obtain technical economy by identifying automata with their behaviors.

*Objects* definable in Algol are similar to automata, but with one difference. The operations of an object have both input and output information. This is in contrast to the instructions of an automaton (the elements of  $M$ ) which are to be regarded as having only input information. The input and output parts of an object operation can be causally interlinked in a complex fashion. So, stream-lined constructions like Mealy machines will not do.

We use coherent spaces to treat the complex input-output breakdown of the object operations. We equip a monoid with a *consistency relation* that we conceptualize in intuitive terms as follows. For elements  $x, y \in M$ , we say that  $x$  and  $y$  are *consistent* and write  $x \circ y$  if  $x$  and  $y$  have differing input information or have the same output information. The complement relation  $x \smile y \iff \neg(x \circ y)$  signifies the opposite, while the *inconsistency* relation  $x \succsim y \iff x \smile y \vee x = y$  signifies that  $x$  and  $y$  have the *same* input information. Suppose  $\alpha : Q \times M \rightarrow Q$  is the transition function of an object. Whenever  $\alpha(q, x)$  is defined, we expect that the output part of  $x$ , as well the final state  $\alpha(q, x)$ , is uniquely determined by  $q$  and the input part of  $x$ . In other words,

$$\alpha(q, x) \text{ and } \alpha(q, y) \text{ are both defined} \implies x \circ y \tag{A.2}$$

(For example, for an active integer object, we define that two distinct integers are always inconsistent. This ensures that  $\alpha(q, i)$  is defined for at most one  $i$ , which is then regarded as the “output” of the object in the state  $q$ .) Suppose  $x = x_1x_2$  and  $y = y_1y_2$  in (A.2) above. Condition (A.1) shows that  $\alpha(q, x_1)$  and  $\alpha(q, x_2)$  are both defined. So, we expect  $x_1 \circ y_1$ . Secondly, if  $x_1 = y_1$  then  $\alpha(\alpha(q, x_1), x_2)$  and  $\alpha(\alpha(q, x_1), y_2)$  are both defined. So, we expect  $y_1 \circ y_2$ . This motivates the basic definition of an object space.

**Definition A.1** *An object space is a pair  $X = (|X|, \circ_X)$  where  $|X| = (|X|, \cdot, e_X)$  is a monoid and  $\circ_X$  is a reflexive-symmetric binary relation on  $|X|$  such that*

$$x_1x_2 \circ_X y_1y_2 \implies x_1 \circ_X y_1 \wedge (x_1 = y_1 \implies x_2 \circ_X y_2)$$

Then  $\dagger A$  creates the “free object space” associated with  $A$ .

Finally, we can regard an *object* for an object space  $X$  as a pair  $(Q, \alpha : Q \times |X| \rightarrow Q)$  satisfying the condition (A.2). The behavior  $L_\alpha(q)$ , for any state  $q \in Q$ , is a left-closed, pairwise-consistent set.

Regular maps  $f : A \rightarrow B$  determine functions from  $A$ -objects  $(Q, \alpha)$  to  $B$ -objects  $(Q, \beta)$ . The transition map  $\beta : Q \times |B| \rightarrow Q$  is given by  $(q, y, q') \in \beta \iff \exists x. x \mapsto y \in f \wedge (q, x, q') \in \alpha$ . (This is the formalization of “simulation” mentioned in Section 3.) Conversely, all functions from

$A$ -objects to  $B$ -objects that are uniform in state sets  $Q$  (in an appropriate sense) arise from regular maps in this fashion.

While “objects” as considered here suffice for the treatment of Algol-like languages, one would want additional structure to treat other features of object-oriented languages such as references, comparison operations and the notion of “self.”

## References

- [1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, August 1978.
- [2] G. Berry. Stable models of typed lambda calculi. In G. Ausiello and C. Boehm, editors, *Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 72–89, Berlin, 1978. Springer-Verlag.
- [3] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. Technical Report 93-219, Rice University, December 1993. To appear in *Information and Computation*, 1994.
- [4] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhauser, Boston, 2 edition, 1993.
- [5] J.-Y. Girard, Y. Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [6] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.
- [7] C.B. Jones. An object-based design method for concurrent programs. Univ of Manchester CS tech report, UMCS-92-12-1, 1992.
- [8] G.M. Kelly. *Basic Concepts of Enriched Category Theory*. Cambridge University Press, 1982. London Math. Soc. Lecture Notes Series, 64.
- [9] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 191–203. ACM, New York, 1988.
- [10] E. Moggi. Computational lambda-calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, 1989. IEEE Computer Society Press.
- [11] M. Odersky. A functional theory of local names. In *Conf. Record 20th ACM Symp. on Principles of Programming Languages*, Charleston, South Carolina, 1993. ACM, New York.
- [12] P. W. O’Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3:435–465, 1993.
- [13] P. W. O’Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. in this volume, 1995.

- [14] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, part 2. *Information and Computation*, 107(1):25–57, 1993.
- [15] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 1995. To appear. Preliminary version appeared in *Conf. Record 20th ACM Symp. on Principles of Programming Languages*, Charleston, South Carolina, pages 171–184. ACM, New York, 1993.
- [16] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.
- [17] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, Cambridge, England, 1985.
- [18] A. Pitts and I. Stark. On the observable properties of higher-order functions that dynamically create local names (preliminary report). In *ACM SIGPLAN Workshop on State in Programming Languages*, pages 31–45, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
- [19] A. M. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science*, number 711 in Lecture Notes in Computer Science, pages 122–141. Springer-Verlag, 1993.
- [20] G. D. Plotkin. Type theory and recursion. 1993.
- [21] U. S. Reddy. Passivity and independence. In *Proceedings, 9th Annual IEEE Symposium on Logic in Computer Science*, pages 342–352. IEEE Computer Society Press, Los Alamitos, California, 1994.
- [22] U. S. Reddy. Global states considered unnecessary: Introduction to object-based semantics. To appear in *Lisp and Symbolic Computation* special issue on state in programming languages, 1995.
- [23] J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978. ACM, New York.
- [24] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.
- [25] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.
- [26] K. Sieber. Full abstraction for the second order subset of an Algol-like language (preliminary report). Technischer Bericht A 01/94, Universitaet des Saarlandes, February, 1994.
- [27] I. A. Stark. Categorical models of local names. Submitted to *Lisp and Symbolic Computation*, special issue on state in programming languages., 1994.

- [28] C. Strachey. *Fundamental Concepts in Programming Languages*. Unpublished lecture notes, International Summer School in Computer Programming, Copenhagen, August 1967.
- [29] R. D. Tennent. Functor-category semantics of programming languages and logics. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 206–224, Guildford, U.K., 1986. Springer-Verlag, Berlin.
- [30] R. D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, 1990.
- [31] R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, 1991.
- [32] D. Walker. Objects in the  $\pi$ -calculus. *Information and Computation*, 116(2):253–271, 1 February 1995.
- [33] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer-Verlag, 1987.