Game Semantics

Samson Abramsky
University of Edinburgh
Department of Computer Science
James Clerk Maxwell Building
Edinburgh EH9 3JZ
Scotland
email: `samson@dcs.ed.ac.uk`

Guy McCusker
St John's College
Oxford OX1 3JP
England
email: `mccusker@comlab.ox.ac.uk`

1 Introduction

The aim of this chapter is to give an introduction to some recent work on the application of game semantics to the study of programming languages.

An initial success for game semantics was its use in giving the first syntax-free descriptions of the fully abstract model for the functional programming language **PCF** [1, 14, 31].

One goal of semantics is to characterize the "universe of discourse" implicit in a programming language or a logic. Thus for a typed, higher-order functional programming language such as **PCF**, one may try to characterize "what it is to be a **PCF**-definable functional". Well established domain-theoretic models [12, 35] provide sufficiently rich universes of functionals to interpret languages such as **PCF**, but in fact they are *too* rich; they include functionals, even "finitary" ones (defined over the booleans, say), which are *not* definable in **PCF**. Moreover, by a remarkable recent result of Ralph Loader [25], this is not an accident; this result (technically the undecidability of observation equivalence on finitary **PCF**) implies that *no* effective characterization of which functionals are definable in **PCF** (even in finitary **PCF**) can exist. Thus in particular a model containing all and only the **PCF**-definable functionals cannot be effectively presentable.

However, rather than focussing on the functionals *in extenso*, we may instead seek to characterize those *computational processes* which arise in computing the functionals. For a sequential, deterministic language such as **PCF** (and most func-

$$\begin{tikzcd}
\mathbf{PCF}\text{+control} \arrow[r, dash] \arrow[d, dash] & \mathbf{PCF}\text{+state+control} \arrow[d, dash] \\
\mathbf{PCF} \arrow[r, dash] & \mathbf{PCF}\text{+state}
\end{tikzcd}$$

Figure 1: The syntactic square

tional languages) these processes should themselves be sequential and deterministic. Indeed, "sequentiality" and "determinacy" are really properties of the *processes*, rather than the functionals they compute, in the first place. However, obtaining an exact characterization along these lines is not easy. One main problem is to avoid unwanted uses of "intensionality", whereby a process computing a functional $F(f)$ can observe properties of the process computing its argument f, rather than only the extensional properties of the function f, and make its output depend on these properties. For this reason, attempts by Kleene [16–23] in his work on higher-type recursion theory, and by Berry and Curien [8] in their work on sequential algorithms, failed to yield a characterization. Similarly, while there were encodings of the λ-calculus into various process calculi such as the π-calculus [29], there was no characterization of which processes arose from these encodings.

The more refined tools provided by game semantics led to a solution of this characterization problem, first in the case of the multiplicative fragment of linear logic [4] and then for **PCF** [1, 14, 31]. Subsequently the first author mapped out a programme of using game semantics to explore the space of programming languages, based on the idea of the "semantic cube". In the present paper, which shall confine our discussion to two dimensions—a semantic square.

Consider first the "syntactic square" of extended typed λ-calculi as shown in Figure 1. The "origin" of this square is occupied by a purely functional language (in this case, **PCF**). Each "axis" corresponds to the extension of the purely functional language by some non-functional feature; those shown in Figure 1 are state (imperative variables) and control operators. (Other possible "axes" include non-determinism and concurrency.) Corresponding to this syntactic square, there is a semantic square of various categories of games and strategies, as shown in Figure 2. The origin of the semantic square is occupied by the category of highly constrained strategies which correspond to the discipline of purely functional programming. The constraints shown in Figure 2 are *innocence* (i) and *bracketing* (b). (These terms will be defined later.) Each axis of the semantic square corresponds to the *relaxation* of one of these constraints on strategies, leading to a larger category. Thus for example $\mathcal{G}_b$ is the category of well-bracketed but not necessarily innocent strategies. Remarkably, there is a precise correspondence between the syntactic and semantic squares, as shown in a series of papers [3,5,7,24]. For example, relaxing the constraint of innocence allows local state to be modelled, while relaxing bracketing allows control

$$\begin{tikzcd}
\mathcal{G}_i \arrow[r, dash] \arrow[d, dash] & \mathcal{G} \arrow[d, dash] \\
\mathcal{G}_{ib} \arrow[r, dash] & \mathcal{G}_b
\end{tikzcd}$$

Figure 2: The semantic square

operators to be modelled. Moreover, these increments in expressive power are *exact*, as shown by *factorization theorems*. For example, every strategy in $\mathcal{G}_b$ can be factored as the composition of (the strategy modelling) a memory cell and an innocent strategy. This immediately reduces definability of **PCF**+state programs in $\mathcal{G}_b$ to that of **PCF** programs in $\mathcal{G}_{ib}$, which was exactly the result obtained in [14,31]. Thus factorization theorems allow the results originally obtained for **PCF** to be transferred to a much richer class of languages incorporating non-functional features. Moreover, as we go beyond the purely functional languages, Loader's result no longer applies, and indeed the game semantics models have been used in a number of cases to yield the first (and still the only) effective constructions of the fully abstract model (see e.g. [5]).

The main body of this paper gives a detailed introduction to these results on **PCF** and its extensions with state and control. The current state of the art has taken matters considerably further, covering recursive types [27], call-by-value [6], and general reference types [3]. Thus all the main features of languages such as Scheme [15] and Core ML [30] have, in principle at least, been accounted for. Current work is addressing a further range of features including concurrency, non-determinism, subtyping and control of interference.

2 Game Semantics: an informal introduction

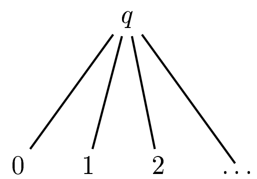
Before proceeding to a detailed technical account in the next section, we will give an informal presentation of the main ideas through examples, with the hope of conveying how close to programming intuitions the formal model is.

As the name suggests, game semantics models computation as the playing of a certain kind of game, with two participants, called Player (P) and Opponent (O). P is to be thought of as representing the system under consideration, while O represents the environment. In the case of programming languages, the system is a term (a piece of program text) and the environment is the context in which the term is used. This is the main point at which games models differ from other process models: the distinction between the actions of the system and those of its environment is made explicit from the very beginning.

In the games we shall consider, O always moves first—the environment sets the system going—and thereafter the two players make moves alternately. What these moves are, and when they may be played, are determined by the rules of each particular game. Since in a programming language a *type* determines the kind of computation which may take place, types will be modelled as games; a *program* of type A determines how the system behaves, so programs will be represented as *strategies* for P, that is, predetermined responses to the moves O may make.

2.1 Modelling Values

In standard denotational semantics, values are *atomic*: a natural number is represented simply as $n \in \omega$. In game semantics, each number is modelled as a simple interaction: the environment starts the computation with an initial move q (a *question*: "What is the number?"), and P may respond by playing a natural number (an *answer* to the question). So the game $\mathbf{N}$ of natural numbers looks like this:



and the strategy for $\mathsf{3}$ is "When O plays q, I will play 3."

$\mathbf{N}$	
q	O
3	P

In diagrams such as the above, time flows downwards: here O has begun by playing q, and at the next step P has responded with 3, as the strategy dictates.

2.2 Functions

The interactions required to model functions are a little more complex. The view taken in game semantics is that the environment of a function consumes the output and provides the input, while the function itself consumes the input and produces the output. The game $\mathbf{N} \Rightarrow \mathbf{N}$ is therefore formed from "two copies of $\mathbf{N}$", one for input, one for output. In the output copy, O may demand output by playing the move q and P may provide it. In the input copy, the situation is reversed: P may demand input with the move q. Thus the O/P role of moves in the input copy is reversed. Plays of this game take the following form.

$\mathbf{N}$	$\Rightarrow$	$\mathbf{N}$	
		q	O
q			P
3			O
		4	P

The play above is a particular run of the strategy modelling the successor function:

> "When O asks for output, I will ask for input; when O provides input n, I will give output $n + 1$."

It is important to notice that the play in each copy of $\mathbf{N}$ (that is, each column of the above diagram) is indeed a valid play of $\mathbf{N}$: it is not possible for O to begin with the third move shown above, supplying an input to the function immediately. Notice also that non-strict functions can be modelled. Here is the strategy which returns 3 without ever investigating what its argument is.

$$\begin{array}{cccc}
\mathbf{N} & \Rightarrow & \mathbf{N} & \\
 & & q & \mathsf{O} \\
 & & 3 & \mathsf{P}
\end{array}$$

These relatively simple ideas let us model all first-order functions. For example, a play in the strategy for addition might look like this.

$$\begin{array}{cccccc}
\mathbf{N} & \Rightarrow & \mathbf{N} & \Rightarrow & \mathbf{N} & \\
 & & & & q & \mathsf{O} \\
q & & & & & \mathsf{P} \\
3 & & & & & \mathsf{O} \\
 & & q & & & \mathsf{P} \\
 & & 2 & & & \mathsf{O} \\
 & & & & 5 & \mathsf{P}
\end{array}$$

The same idea lets us form $A \Rightarrow B$ for any games A and B: take a copy of A and a copy of B, "place them side by side" and reverse the O/P roles of the moves in A.

2.3 Higher-order functions

The strategy for the function $\lambda f.\ f\ 0\ 1$ plays as follows.

$$\begin{array}{cccccccc}
(\mathbf{N} & \Rightarrow & \mathbf{N} & \Rightarrow & \mathbf{N}) & \Longrightarrow & \mathbf{N} & \\
 & & & & & & q & \mathsf{O} \\
 & & & & q & & & \mathsf{P} \\
q & & & & & & & \mathsf{O} \\
0 & & & & & & & \mathsf{P} \\
 & & q & & & & & \mathsf{O} \\
 & & 1 & & & & & \mathsf{P} \\
 & & & & n & & & \mathsf{O} \\
 & & & & & & n & \mathsf{P}
\end{array}$$

Here O plays the role of the function f in the game $(\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N})$ as well as demanding output from the rightmost $\mathbf{N}$. P first asks for the output from f; when O asks for the first input to f, P supplies 0; when O asks for the second input, P supplies 1; and when O supplies n as output from f, P copies this as the overall output.

The choice of moves made by O in the example above is by no means the only one. For example, O could ask for the arguments to f in the other order, or could neglect to ask for the arguments at all. But P's strategy would be the same regardless: answer 0 to the first input, 1 to the second, and copy the output of f as the overall output.

Higher-order functions in general use their arguments more than once. For example, the strategy for function $\lambda f.\ f(0) + f(1)$ needs to play out two interactions with its input f:

$$
\begin{array}{ccccccl}
(\mathbf{N} & \Rightarrow & \mathbf{N}) & \Longrightarrow & \mathbf{N} & & \\
 & & & & q & & \mathsf{O} \\
 & & q & & & & \mathsf{P} \\
q & & & & & & \mathsf{O} \\
0 & & & & & & \mathsf{P} \\
 & & n & & & & \mathsf{O} \\
 & & q & & & & \mathsf{P} \\
q & & & & & & \mathsf{O} \\
1 & & & & & & \mathsf{P} \\
 & & m & & & & \mathsf{O} \\
 & & & & n+m & & \mathsf{P}
\end{array}
$$

The play on the left here is not a single run of $\mathbf{N} \to \mathbf{N}$ but rather two such runs, one after another. It is also possible for runs to be interleaved. For example, $\lambda f.\ f(f(3))$ plays thus:

$$
\begin{array}{ccccccl}
(\mathbf{N} & \Rightarrow & \mathbf{N}) & \Longrightarrow & \mathbf{N} & & \\
 & & & & q & & \mathsf{O} \\
 & & q & & & & \mathsf{P} \\
q & & & & & & \mathsf{O} \\
 & & q & & & & \mathsf{P} \\
q & & & & & & \mathsf{O} \\
3 & & & & & & \mathsf{P} \\
 & & n & & & & \mathsf{O} \\
n & & & & & & \mathsf{P} \\
 & & m & & & & \mathsf{O} \\
 & & & & m & & \mathsf{P}
\end{array}
$$

Here P's first action is to ask about the output of f. When f (played by O) asks for input, P again asks for output from f, since the function in question supplies the output of $f(3)$ as input to the outermost call of f. When O now asks for input, P can supply 3. O then plays some output n, which represents the value $f(3)$, so P copies n as the input to the first call of f. The output m then represents $f(f(3))$ so P copies it as the overall output.

2.4 A difficulty

As we have seen, the play on the left hand side of a game $A \Rightarrow B$ may consist of several plays of A, interleaved in chunks of even length. If we represent these interleaved sequences as we have been doing, using just the "underlying" sequences of moves without regard to how the different chunks should be pasted together to form individual plays of A, then our model does not in fact carry enough information to model higher-order functions correctly. To see this, it is unfortunately necessary to consider an example of a rather high type, $((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$.

Consider the following λ-terms.

$$
\begin{array}{rcl}
M_1 & = & \lambda f.\, f(\lambda x.\, f(\lambda y.\, y)) \\
M_2 & = & \lambda f.\, f(\lambda x.\, f(\lambda y.\, x))
\end{array}
$$

In the semantics outlined above, each of these determines the following strategy.

$$
\begin{array}{ccccccc}
((\mathbf{N} & \Rightarrow & \mathbf{N}) & \Rightarrow & \mathbf{N}) & \Rightarrow & \mathbf{N} \\
 & & & & & & q \\
 & & & & q & & \\
 & & q & & & & \\
 & & & & q & & \\
 & & q & & & & \\
q & & & & & & \\
 & & & \vdots & & &
\end{array}
$$

What is going on here? Let us consider each O-move and P's response in turn.

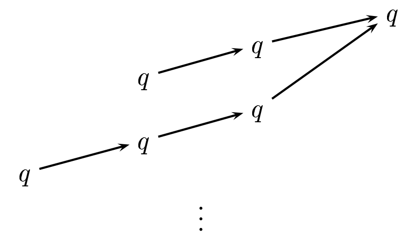
1. O begins by asking for output. The output will be the result of applying f to an argument, so P demands output from f.

2. f asks about its first input, which will be some function $g = \lambda x. \dots$ of type $\mathbf{N} \Rightarrow \mathbf{N}$. At this point P *could* ask what input O will provide, i.e. P could ask about x; but P knows that the output of the function g comes from f, so P again requests output from f.

3. Again, f asks for its input, which will be some function $g' = \lambda y. \dots$. This time P asks for an input to a function of type $\mathbf{N} \Rightarrow \mathbf{N}$ which is itself an input to f. But is this the function g or g'? That is, is P asking about x or about y? The final move here is ambiguous, because it could form part of either of two runs of $(\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$ which are being interleaved. As a result, the terms M_1 and M_2 are identified, which they should not be.

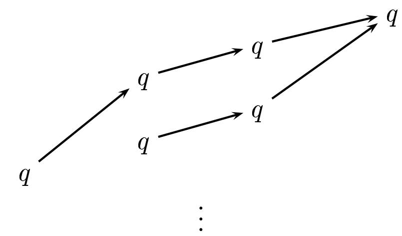
The solution to this problem is to disambiguate such moves by tying the various interleaved runs together in an appropriate way. The path we take is to attach *pointers* to moves: along with each move comes a pointer to a previous move, which determines the copy of a sub-game to which the move belongs. The above examples become:

$$((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$$



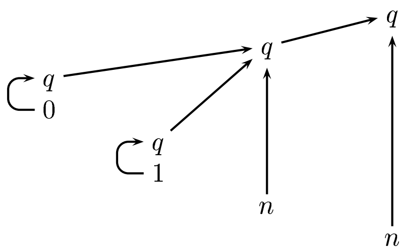
for M_1, and

$$((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$$



for M_2. Each request for input to an occurrence of function carries a pointer to the move which demanded output from that occurrence. Pointers are also attached to answers, pointing back to the question being answered, so that our earlier example $\lambda f.\ f\ 0\ 1$ becomes:

$$(\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}) \Longrightarrow \mathbf{N}$$



2.5 Products

Another way of forming new types from old is to take products: the type $A \times B$ consists of pairs of elements, one of type A and one of type B. In game semantics this is handled in a similar way to the function space constructor: the game $A \times B$ is a copy of A and a copy of B, side by side. When O begins, he decides whether to play in A or in B, and then plays a move there. From then on, a play of either A or B is carried out. A *strategy* for $A \times B$ therefore determines a strategy for A and one for B, according to how O begins, so indeed corresponds to an appropriate pair of strategies. For instance, the strategy corresponding to the pair $\langle 3, 5\rangle$ has the following two plays:

$$
\begin{array}{cccl}
\mathbf{N} & \times & \mathbf{N} & \\
q & & & \mathsf{O} \\
3 & & & \mathsf{P} \\
\cline{1-3}
& & q & \mathsf{O} \\
& & 5 & \mathsf{P}
\end{array}
$$

Notice that in any given play, only one side of the product is investigated; so, if a function of type $\mathbf{N} \times \mathbf{N} \Rightarrow \mathbf{N}$ wishes to use both components of the input, it must interrogate its input twice. For example, the strategy for addition can be given this type, becoming the following.

$$
\begin{array}{ccccccl}
\mathbf{N} & \times & \mathbf{N} & \Rightarrow & \mathbf{N} & \\
& & & & q & \mathsf{O} \\
q & & & & & \mathsf{P} \\
3 & & & & & \mathsf{O} \\
& & q & & & \mathsf{P} \\
& & 2 & & & \mathsf{O} \\
& & & & 5 & \mathsf{P}
\end{array}
$$

Here we have turned a strategy for $\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ into one for $\mathbf{N} \times \mathbf{N} \Rightarrow \mathbf{N}$. The same can be done for any strategy of type $A \Rightarrow B \Rightarrow C$, and can of course be reversed as well: the familiar operation of (un)currying from functional programming becomes a trivial realignment of moves in game semantics.

Let us also take the time to mention the nullary version of the product: the empty game, which we write as I. This game has no moves, so that the plays of $I \Rightarrow A$ are the same as those of A (hence strategies for A and strategies for $I \Rightarrow A$ are the same). Note also that there is no way to play a move in the game $A \Rightarrow I$.

2.6 Interaction: composition of strategies

Game semantics is intended to provide a *compositional* interpretation of programs: just as small programs can be put together to form large ones, so strategies can be combined to form new strategies. The fundamental "glue" in traditional denotational semantics is function application; for game semantics it is *interaction* of strategies which gives us a notion of composition.

Consider the strategy for addition, with the type $\mathbf{N} \times \mathbf{N} \Rightarrow \mathbf{N}$. In order to compose this with the strategy $\langle 3, 5 \rangle : I \Rightarrow \mathbf{N} \times \mathbf{N}$, we let the two strategies interact with one another. When `add` plays a move in $\mathbf{N} \times \mathbf{N}$, we feed it as an O-move to $\langle 3, 5 \rangle$; conversely, when this strategy plays in $\mathbf{N} \times \mathbf{N}$, we feed this move as an O-move back to `add`.

$$
\begin{array}{ccccccc}
 & \langle 3,5\rangle & & & & \texttt{add} & \\
I & \Longrightarrow & \mathbf{N} & \times & \mathbf{N} & \Longrightarrow & \mathbf{N} \\
 & & & & & & q \\
 & & q & & & & \\
 & & 3 & & & & \\
 & & & & q & & \\
 & & & & 5 & & \\
 & & & & & & 8
\end{array}
$$

By hiding the action in the middle game, we obtain the strategy

$$
\begin{array}{ccc}
I & \Rightarrow & \mathbf{N} \\
 & & q \\
 & & 8
\end{array}
$$

representing the number 8 as expected. So in game semantics, composition of functions is modelled by CSP-style "parallel composition + hiding" [13].

Notice that in the above composition, the strategy for addition calls on the pair $\langle 3, 5\rangle$ twice; other strategies could have called upon it more than twice. In a composition such as this, the strategy on the left hand side may be called repeatedly. It is a property of functional programs that no matter how often they are called and how they are used in each call, they behave the same way each time. This property is a consequence of the condition of *innocence* which we will impose on our strategies when modelling functional programming.

2.7 Copycat strategies

For any game A, there is a strategy on $A \Rightarrow A$ which responds to any O-move in one copy of A by playing the same move in the other copy, where it will be a P-move, thanks to the reversal of roles. For example, in $\mathbf{N} \Rightarrow \mathbf{N}$, we get the strategy which represents the identity function.

$$
\begin{array}{ccc}
\mathbf{N} & \Rightarrow & \mathbf{N} \\
 & & q \\
q & & \\
n & & \\
 & & n
\end{array}
$$

We refer to such strategies as *copycat* strategies. The copycat strategy on each $A \Rightarrow A$ is an identity for the composition defined above. For example, consider composing the copycat strategy (on the left) with the strategy for $\lambda f.\ f(3)$ (on the right).

$$
\begin{array}{ccccccccc}
 & & & \text{copycat} & \cdots & \cdots & \lambda f.\ f(3) & & \\
(\mathbf{N} & \Rightarrow & \mathbf{N}) & \Longrightarrow & (\mathbf{N} & \Rightarrow & \mathbf{N}) & \Longrightarrow & \mathbf{N} \\
 & & & & & & & & q \\
 & & & & & & q & & \\
 & & q & & & & & & \\
q & & & & & & & & \\
 & & & & q & & & & \\
 & & & & 3 & & & & \\
3 & & & & & & & & \\
 & & n & & & & & & \\
 & & & & & & n & & \\
 & & & & & & & & n
\end{array}
$$

After hiding, we are left with exactly the same play in the outer games as we see in the middle and right games, that is, the play of $\lambda f.\ f(3)$.

2.8 Imperative Languages: commands

The fundamental operations of imperative languages are commands. In game semantics we take the view that commands inhabit a "unit type" or "void type", as in Standard ML or Java. The game `com` is extremely simple:

$$
\begin{array}{c}
\mathtt{run} \\
| \\
\mathtt{done}
\end{array}
$$

This can be thought of as a kind of "scheduler interface": the environment of a command has the opportunity to schedule it by playing the move `run`. When the command is finished, it returns control to the environment by playing `done`.

We can interpret the "do nothing" command `skip` very easily as the strategy which responds to `run` with `done` immediately. The following strategy interprets sequential composition.

$$
\begin{array}{cccccc}
\mathtt{seq}: & \mathtt{com} & \Rightarrow & \mathtt{com} & \Rightarrow & \mathtt{com} \\
 & & & & & \mathtt{run} \\
 & \mathtt{run} & & & & \\
 & \mathtt{done} & & & & \\
 & & & \mathtt{run} & & \\
 & & & \mathtt{done} & & \\
 & & & & & \mathtt{done}
\end{array}
$$

This can be thought of as a scheduler: when activated, it first schedules its first argument, and waits for that to complete before scheduling the second argument. When that is complete, the whole sequential composition is complete.

2.9 Store

The interesting part of an imperative language is of course the store upon which the commands operate. To interpret mutable variables, we will take an "object-oriented view" as advocated by John Reynolds [34]. In this view, a variable is seen as an object with two methods:

- the "read method", for dereferencing, giving rise to an operation of type $\mathtt{var} \Rightarrow \mathbf{N}$;
- the "write method", for assignment, giving an operation of type $\mathtt{var} \Rightarrow \mathbf{N} \Rightarrow \mathtt{com}$.

We *identify* the type of variables with the product of the types of these methods, setting

$$\mathtt{var} = (\mathbf{N} \Rightarrow \mathtt{com}) \times \mathbf{N}.$$

Now assignment and dereferencing are just the two projections, and we can interpret a command `x:=!x+1` as the strategy

$$\begin{array}{ccccccc}
(\mathbf{N} & \Rightarrow & \mathtt{com}) & \times & \mathbf{N} & \Longrightarrow & \mathtt{com} \\
 & & & & & & \mathtt{run} \\
 & & & & \mathtt{read} & & \\
 & & & & n & & \\
 & & \mathtt{write} & & & & \\
q & & & & & & \\
n+1 & & & & & & \\
 & & \mathtt{ok} & & & & \\
\end{array}$$

(We use `write` and `ok` in place of `run` and `done` in the assignment part, and `read` in place of q in the dereferencing part, to emphasize that these moves initiate assignments and dereferencing rather than arbitrary commands or natural number expressions.)

The vital thing is to interpret the *allocation* of variables correctly, so that if the variable `x` in the above example has been bound to a genuine storage cell, the various reads and writes made to it have the expected relationship. In general, a term M with a free variable x will be interpreted as a strategy for $\mathtt{var} \Rightarrow A$, where A is the type of M. We must interpret $\mathtt{new}\ x\ \mathtt{in}\ M$ as a strategy for A by "binding x to a memory cell". With game semantics, this is easy! The strategy for M will play some moves in A, and may also make repeated use of the `var` part. The play in the `var` part will look something like this.

$$
\begin{array}{ccccc}
(\mathbf{N} & \Rightarrow & \mathtt{com}) & \times & \mathbf{N} \\
 & & \mathtt{write} & & \\
q & & & & \\
3 & & & & \\
 & & \mathtt{ok} & & \\
 & & \mathtt{write} & & \\
q & & & & \\
5 & & & & \\
 & & \mathtt{ok} & & \\
 & & & & \mathtt{read} \\
 & & & & 6 \\
 & & & & \mathtt{read} \\
 & & & & 7 \\
 & & & \vdots &
\end{array}
$$

Of course there is nothing constraining the reads and writes to have the expected relationship. However, there is an obvious strategy

$$\mathtt{cell} : I \Rightarrow \mathtt{var}$$

which plays like a storage cell, always responding to a `read` with the last value written. Once we have this strategy, we can interpret `new` by composition with `cell`, so $[\![\mathtt{new}\ x\ \mathtt{in}\ M]\!]$ is

$$I \overset{\mathtt{cell}}{\Longrightarrow} \mathtt{var} \overset{[\![M]\!]}{\Longrightarrow} A$$

Two important properties of local variables are immediately captured by this interpretation:

Locality Since the action in `var` is hidden by the composition, the environment is unaware of the existence and use of the local variable.

Irreversibility As M interacts with `cell`, there is no way for M to undo any writes which it makes. Of course M can return the value stored in the cell to be the same as it has earlier been, but only by performing a new `write`.

However, it is vital to the correct interpretation of variables that when composing M with `cell`, the behaviour of `cell` *varies from one call to the next*, depending on what writes are made: each read or write constitutes a complete call of the `cell` strategy, and of course any write affects subsequent reads. Such behaviour does not arise in a functional program, and in fact means that `cell` *violates the condition of "innocence"*.

A similar approach to modelling variables has also been taken in process models, for example by Milner in CCS [28].

2.10 Control operators

Let us consider a very simple control operator which allows early escapes from the evaluation of functions. The constructor

$$\mathtt{catch} : (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$$

has the property that

$$\mathtt{catch}\ f = \begin{cases} 0 & \text{if } f \text{ calls its argument.} \\ n+1 & \text{if } f \text{ returns } n \text{ without calling its argument.} \end{cases}$$

Such an operator can be defined in Scheme or SML/NJ from `call/cc`, for example.

The corresponding strategy has the following two forms of play.

$$(i) \quad \begin{array}{ccccc} (\mathbf{N} & \Rightarrow & \mathbf{N}) & \Rightarrow & \mathbf{N} \\ & & & & q \\ & & q & & \\ q & & & & \\ & & & & 0 \end{array} \qquad (ii) \quad \begin{array}{ccccc} (\mathbf{N} & \Rightarrow & \mathbf{N}) & \Rightarrow & \mathbf{N} \\ & & & & q \\ & & q & & \\ & & n & & \\ & & & & n+1 \end{array}$$

In computation (i) there are "dangling questions" left when the initial question has been answered—something which never happened in any of our previous examples. In fact, this is symptomatic of the fact that this strategy *violates the "bracketing condition"*.

2.11 A semantic characterization of programming disciplines

We have seen that it is possible to interpret functional programs, programs with store and programs with control operators using game semantics. However, we have also noticed that the behaviour of functional programs obeys certain principles which may be violated by programs written in more expressive languages. In fact, the analysis provided by game semantics allows us to give an answer to the question

> "What is it to be a functional computational process?"

The answer has three elements.

Determinacy In any given position, what the strategy does is uniquely determined.

Innocence What the strategy does is in fact determined by *partial* information about the history—the local "functional context" of our subcomputation. In particular, an innocent strategy cannot vary its behaviour from one call to the next.

Bracketing Control flow follows a properly nested call-return discipline.

These are simple, local constraints on strategies. Remarkably, they suffice to characterize our space of programming languages.

Constraints	Language
D + I + B	purely functional
D + I	functional + control
D + B	functional + store
D	functional + store + control

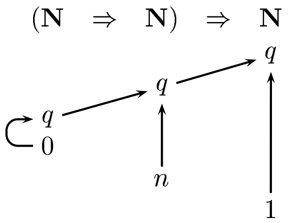
2.12 Innocence

We conclude this informal introduction with an illustration of how innocence constrains strategies, in the form of the following non-innocent example. Consider the program

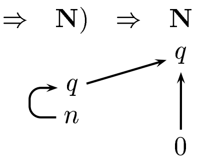
$$\lambda f : \mathbf{N} \Rightarrow \mathbf{N}.\ \ \texttt{new}\ x := 0\ \texttt{in}$$
$$\texttt{ifzero}\ f(x := 1;\ \texttt{return}\ 0)\ \texttt{then}\ !x\ \texttt{else}\ !x.$$

(Here `ifzero` is used simply to force evaluation of $f(x := 1;\ \texttt{return}\ 0)$ in this lazy language.) This program takes an argument f, and evaluates $f(0)$, but records the fact that f uses its argument by means of a side-effect to x. After f returns, the function returns 1 if f used its argument, and 0 otherwise. (This behaviour is quite similar to that of `catch`, but in this case f must return a value if the overall function is to return a value.)

The corresponding strategy has two plays of interest to us.



and



A strategy is innocent if its response at a given position depends only on the local "functional context" of the position, rather than the entire history. This functional context, called the *view*, is calculated by removing moves surrounded by pairs $m \ldots n$ where n is an O-move.

$$\ulcorner s \cdot m \cdot t \cdot n \urcorner = \ulcorner s \urcorner \cdot m \cdot n.$$

So after the move n in each of the diagrams above, the view is just $q \cdot q \cdot n$: the information about whether f uses its argument or not is *not* part of the view. This confirms the fact that no purely functional program has the behaviour of this imperative program.

3 Categories of Games

We now begin a more technical development of the ideas outlined earlier by giving formal definitions of the categories of games and strategies in which programming languages are modelled. The definitions given here are taken from [27], but are essentially adaptations of the original definitions given by Hyland and Ong [14], taking into account the ideas of Abramsky, Jagadeesan and Malacaria [1], particularly with regard to the linear type structure of the categories. Similar games, and in fact the very same model of the language **PCF**, were also discovered by Nickau [31].

We shall describe eight different categories of games. The basic categories are $\mathcal{G}$, which is (almost) a model of the $(!, \otimes, \multimap, \&)$ fragment of intuitionistic linear logic, and $\mathcal{C}$, a cartesian closed category built out of $\mathcal{G}$ using the Girard translation of intuitionistic logic into linear logic: $A \Rightarrow B = !A \multimap B$ [11]. The morphisms of both these categories are strategies. By putting constraints of innocence and well-bracketedness on strategies, we obtain various subcategories both of $\mathcal{G}$ and of $\mathcal{C}$, leading to eight different categories, of which four are models of linear logic, the other four being cartesian closed.

3.1 Arenas, views and legal positions

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

- M_A is a set of *moves*;
- $\lambda_A : M_A \to \{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$ is a *labelling* function which indicates whether a move is by Opponent (O) or Player (P), and whether it is a question (Q) or an answer (A). We write the set $\{\mathsf{O}, \mathsf{P}\} \times \{\mathsf{Q}, \mathsf{A}\}$ as $\{\mathsf{OQ}, \mathsf{OA}, \mathsf{PQ}, \mathsf{PA}\}$, and use λ_A^{OP} to mean λ_A followed by left projection, so that $\lambda_A^{\mathsf{OP}}(m) = \mathsf{O}$ if $\lambda_A(m) = \mathsf{OQ}$ or $\lambda_A(m) = \mathsf{OA}$. Define λ_A^{QA} in a similar way. Finally, $\overline{\lambda}_A$ is λ_A with the O/P part reversed, so that

$$\overline{\lambda}_A(m) = \mathsf{OQ} \iff \lambda_A(m) = \mathsf{PQ}$$

  and so on. If $\lambda^{\mathsf{OP}}(m) = \mathsf{O}$, we call m an O-move; otherwise, m is a P-move;
- $\vdash_A$ is a relation between $M_A + \{\star\}$ (where $\star$ is just a dummy symbol) and M_A, called *enabling*, which satisfies

  (e1) $\star \vdash_A m \Rightarrow \lambda_A(m) = \mathsf{OQ} \wedge [n \vdash_A m \iff n = \star]$;
  (e2) $m \vdash_A n \wedge \lambda_A^{\mathsf{QA}}(n) = \mathsf{A} \Rightarrow \lambda_A^{\mathsf{QA}}(m) = \mathsf{Q}$;
  (e3) $m \vdash_A n \wedge m \neq \star \Rightarrow \lambda_A^{\mathsf{OP}}(m) \neq \lambda_A^{\mathsf{OP}}(n)$.

The idea of the enabling relation is that when a game is played, a move can only be made if a move has already been made to enable it. The $\star$ enabler is special—it says which moves are enabled at the outset. A move m such that $\star \vdash_A m$ is called *initial*. Conditions (e2) and (e3) say that answers are enabled by questions, and that the protagonists always enable each other's moves, never their own.

Given an arena, we are interested in sequences of moves of a certain kind. Before defining these, let us fix our notation for operations on sequences. If s and t are sequences, we write st for their concatenation. We also write sa for the sequence s with element a appended. Sometimes we use the notation $s \cdot t$ or $s \cdot a$ when it aids legibility. The empty sequence is written as ε, and $\sqsubseteq$ denotes the prefix ordering on sequences.

Definition A *justified sequence* in an arena A is a sequence s of moves of A, together with an associated sequence of pointers: for each non-initial move m in s, there is a pointer to a move n earlier in s such that $n \vdash_A m$. We say that the move n *justifies* m. Note that the first move in any justified sequence must be initial, since it cannot possibly have a pointer to an earlier move attached to it; so by (e1), justified sequences always start with an opponent question.

Given a justified sequence s, define the *player view* $\ulcorner s \urcorner$ and *opponent view* $\llcorner s \lrcorner$ of s by induction on $|s|$, as follows.

$$
\begin{array}{rcll}
\ulcorner \varepsilon \urcorner & = & \varepsilon. & \\
\ulcorner s \cdot m \urcorner & = & \ulcorner s \urcorner m, & \text{if } m \text{ is a P-move.} \\
\ulcorner s \cdot m \urcorner & = & m, & \text{if } \star \vdash m. \\
\ulcorner s \cdot m \cdot t \cdot n \urcorner & = & \ulcorner s \urcorner \cdot m \cdot n, & \text{if } n \text{ is an O-move.} \\
\llcorner \varepsilon \lrcorner & = & \varepsilon. & \\
\llcorner s \cdot m \lrcorner & = & \llcorner s \lrcorner m, & \text{if } m \text{ is an O-move.} \\
\llcorner s \cdot m \cdot t \cdot n \lrcorner & = & \llcorner s \lrcorner \cdot m \cdot n, & \text{if } n \text{ is a P-move.}
\end{array}
$$

Notice that the view of a justified sequence need not itself be justified: the appearance of a move m in the view does not guarantee the appearance of its justifier. This will be rectified when we impose the *visibility condition*, to follow.

A justified sequence s is *legal*, or is a *legal position*, if it also satisfies the following *alternation* and *visibility conditions*:

- Players alternate: if $s = s_1 mn s_2$ then $\lambda^{\mathsf{OP}}(m) \neq \lambda^{\mathsf{OP}}(n)$.
- if $tm \sqsubseteq s$ where m is a P-move, then the justifier of m occurs in $\ulcorner t \urcorner$.
- if $tm \sqsubseteq s$ where m is a non-initial O-move, then the justifier of m occurs in $\llcorner t \lrcorner$.

We write L_A for the set of legal positions of A.

3.2 Games and strategies

Definition Let s be a legal position of an arena A and let m be a move in s. We say that m is *hereditarily justified* by an occurrence of a move n in s if the chain of justification pointers leading back from m ends at n, i.e. m is justified by some move m_1, which is in turn justified by m_2 and so on until some m_k is justified by an initial move n. We write $s \upharpoonright n$ for the subsequence of s containing all moves hereditarily justified by n. This notation is slightly ambiguous, because it confuses the move n with a particular occurrence of n; however, no difficulty will arise in practice. We similarly define $s \upharpoonright I$ for a set I of (occurrences of) initial moves in s to be the subsequence of s consisting of all moves hereditarily justified by a move of I.

A *game* A is specified by a structure $\langle M_A, \lambda_A, \vdash_A, P_A \rangle$ where

- $\langle M_A, \lambda_A, \vdash_A \rangle$ is an arena.
- P_A is a non-empty, prefix-closed subset of L_A, called the *valid positions*, and satisfying

  if $s \in P_A$ and I is a set of initial moves of s then $s \upharpoonright I \in P_A$.

3.2.1 Multiplicatives

Given games A and B, define new games $A \otimes B$ and $A \multimap B$ as follows.

$$
\begin{array}{rcl}
M_{A\otimes B} & = & M_A + M_B. \\
\lambda_{A\otimes B} & = & [\lambda_A, \lambda_B]. \\
\star \vdash_{A\otimes B} n & \Longleftrightarrow & \star \vdash_A n \vee \star \vdash_B n. \\
m \vdash_{A\otimes B} n & \Longleftrightarrow & m \vdash_A n \vee m \vdash_B n. \\
P_{A\otimes B} & = & \{s \in L_{A\otimes B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}.
\end{array}
$$

$$
\begin{array}{rcll}
M_{A\multimap B} & = & M_A + M_B. & \\
\lambda_{A\multimap B} & = & [\overline{\lambda}_A, \lambda_B]. & \\
\star \vdash_{A\multimap B} m & \Longleftrightarrow & \star \vdash_B m & \\
m \vdash_{A\multimap B} n & \Longleftrightarrow & m \vdash_A n \vee m \vdash_B n \vee & \\
 & & [\star \vdash_B m \wedge \star \vdash_A n] & \text{for } m \neq \star. \\
P_{A\multimap B} & = & \{s \in L_{A\multimap B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}. &
\end{array}
$$

In the above, $s \upharpoonright A$ denotes the subsequence of s consisting of all moves from M_A; $s \upharpoonright B$ is analogous. The conflict with the previously introduced notation $s \upharpoonright I$ should not cause any confusion.

The tensor unit is defined by $I = \langle \emptyset, \emptyset, \emptyset, \{\varepsilon\} \rangle$.

3.2.2 Strategies

Definition A *strategy* σ for a game A is a non-empty set of even-length positions from P_A, satisfying

(s1) $sab \in \sigma \Rightarrow s \in \sigma$.

(s2) $sab, sac \in \sigma \Rightarrow b = c$, and the justifier of b is the same as that of c. In other words, the justified sequences sab and sac are identical.

The *identity* strategy for a game A is a strategy for $A \multimap A$ defined by

$$\mathsf{id}_A = \{s \in P_{A_1 \multimap A_2} \mid \forall t \sqsubseteq^{\text{even}} s.(t \upharpoonright A_1 = t \upharpoonright A_2)\}.$$

We use subscripts to distinguish the two occurrences of A, and write $t \sqsubseteq^{\text{even}} s$ to mean that t is an even-length prefix of s.

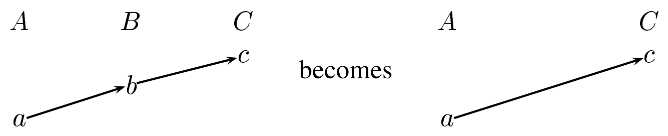
All that id_A does is to copy the move made by Opponent in one copy of A to the other copy of A. The justifier for Player's move is the copy of the justifier of Opponent's move. It is easy to check that this does indeed define a strategy.

3.2.3 Composition

The categories we will work in have games as objects and strategies as morphisms. Therefore, given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, we would like to compose them to form a strategy $\sigma ; \tau : A \multimap C$. First, some auxiliary definitions are necessary.

Definition Let u be a sequence of moves from games A, B and C together with justification pointers from all moves except those initial in C. Define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C; if a pointer from one of these points to a move of A, delete that pointer. Similarly define $u \upharpoonright A, B$. We say that u is an *interaction sequence* of A, B and C if $u \upharpoonright A, B \in P_{A \multimap B}$ and $u \upharpoonright B, C \in P_{B \multimap C}$. The set of all such sequences is written as $\mathsf{int}(A, B, C)$.

Suppose $u \in \mathsf{int}(A, B, C)$. A pointer from a C-move must be to another C-move, and a pointer from an A-move a must be either to another A-move, or to an *initial* B-move, b, which in turn must have a pointer to an initial C-move, c. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C, except that in the case outlined above, the pointer from a is changed to point to c.



Given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, define $\sigma \parallel \tau$ to be

$$\{u \in \mathsf{int}(A, B, C) \mid u \upharpoonright A, B \in \sigma \wedge u \upharpoonright B, C \in \tau\}.$$

We are now ready to define the composite of two strategies.

Definition If $\sigma : A \multimap B$ and $\tau : B \multimap C$, define $\sigma ; \tau : A \multimap C$ by

$$\sigma ; \tau = \{u \upharpoonright A, C \mid u \in \sigma \parallel \tau\}.$$

3.2.4 Constraining strategies

Two classes of strategies will be of special interest: the *innocent* ones and the *well-bracketed* ones.

Definition Given positions $sab, ta \in L_A$, where sab has even length and $\ulcorner sa \urcorner = \ulcorner ta \urcorner$, there is a unique extension of ta by the move b together with a justification pointer in such a way that $\ulcorner sab \urcorner = \ulcorner tab \urcorner$. Call this extension $\mathsf{match}(sab, ta)$. The a strategy $\sigma : A$ is *innocent* if and only if it satisfies

$$sab \in \sigma \wedge t \in \sigma \wedge ta \in P_A \wedge \ulcorner ta \urcorner = \ulcorner sa \urcorner \Rightarrow \mathsf{match}(sab, ta) \in \sigma.$$

In other words, the move and pointer played by an innocent strategy σ at a position sa is determined by the P-view $\ulcorner sa \urcorner$.

A strategy $\sigma : A$ is *well-bracketed* (or *satisfies the bracketing condition*) if and only if for every $sab \in \sigma$ with b an answer, the justification pointers on sab have the form

$$\ldots q \cdot q_1 \ldots a_1 \cdot q_2 \ldots a_2 \ldots q_n \ldots a \cdot b$$

where the moves a_1, a_2, ... and a are all answers. That is to say, when P gives an answer, it is in answer to the most recent unanswered question in the view: we call this the *pending question*. Note that O moves are *not* required to satisfy this condition.

3.2.5 Four categories of games

We now define four categories of games: $\mathcal{G}$, $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}_{ib}$. The objects of all these categories are games. A morphism from A to B in $\mathcal{G}$ is a strategy $\sigma : A \multimap B$. The lluf subcategory $\mathcal{G}_i$ has as morphisms only the innocent strategies; $\mathcal{G}_b$ has only the well-bracketed strategies; and $\mathcal{G}_{ib}$ has only the innocent and well-bracketed strategies. In all cases, composition and identities are as described above.

Proposition 1 $\mathcal{G}$, $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}_{ib}$ are categories.

The content of this proposition is that composition of strategies is well-defined, associative, has the copycat strategy id as unit, and preserves innocence and well-bracketedness. Proofs of these facts can be found in [1, 14, 27].

3.2.6 Monoidal structure

We have already given the object part of the tensor product. We now describe the corresponding action on morphisms which makes tensor into a bifunctor and $\mathcal{G}$ into a symmetric monoidal category.

Given $\sigma : A \to B$ and $\tau : C \to D$, define $\sigma \otimes \tau : (A \otimes C) \to (B \otimes D)$ by

$$\sigma \otimes \tau = \{ s \in L_{A \otimes C \multimap B \otimes D} \mid s \restriction A, B \in \sigma \wedge s \restriction C, D \in \tau \}.$$

We can now define natural isomorphisms unit, assoc and comm with components $\mathsf{unit}_A : A \otimes I \to A$, $\mathsf{assoc}_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C$ and $\mathsf{comm}_{A,B} : A \otimes B \to B \otimes A$ given by the obvious copycat strategies—in each case the set of moves of the domain game is isomorphic to the set of moves of the codomain game. It is then trivial to verify the following.

Proposition 2 The structure described above makes $\mathcal{G}$ into a symmetric monoidal category. $\mathcal{G}_i$ and $\mathcal{G}_b$ are sub-symmetric monoidal categories of $\mathcal{G}$ (that is, they are subcategories which inherit a symmetric monoidal structure from that of $\mathcal{G}$), and $\mathcal{G}_{ib}$ is a sub-symmetric monoidal category of each of $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}$.

3.2.7 Closed structure

To make $\mathcal{G}$ into a symmetric monoidal *closed* category, we need to show that each functor $- \otimes B$ has a (specified) right adjoint. Observe first that the only difference between games $A \otimes B \multimap C$ and $A \multimap (B \multimap C)$ is in the tagging of moves in the disjoint unions. Therefore

$$\begin{array}{rcl}\mathcal{G}(A \otimes B, C) & = & \{\sigma \mid \sigma \text{ is a strategy for } A \otimes B \multimap C\} \\ & \cong & \{\sigma \mid \sigma \text{ is a strategy for } A \multimap (B \multimap C)\} \\ & = & \mathcal{G}(A, B \multimap C).\end{array}$$

This structure gives us:

Proposition 3 $\mathcal{G}$ is an autonomous (i.e. symmetric monoidal closed) category. $\mathcal{G}_i$ and $\mathcal{G}_b$ are sub-autonomous categories of $\mathcal{G}$, and $\mathcal{G}_{ib}$ is a sub-autonomous category of each of $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}$.

3.2.8 Products

Given games A and B, define a game $A\&B$ as follows.

$$\begin{array}{rcl}M_{A\&B} & = & M_A + M_B \\ \lambda_{A\&B} & = & [\lambda_A, \lambda_B] \\ \star \vdash_{A\&B} n & \Longleftrightarrow & \star \vdash_A n \vee \star \vdash_B n \\ m \vdash_{A\&B} n & \Longleftrightarrow & m \vdash_A n \vee m \vdash_B n \\ P_{A\&B} & = & \{s \in L_{A\&B} \mid s \restriction A \in P_A \wedge s \restriction B = \varepsilon\} \\ & \cup & \{s \in L_{A\&B} \mid s \restriction B \in P_B \wedge s \restriction A = \varepsilon\}.\end{array}$$

We can now define projections $\pi_1 : A\&B \to A$ and $\pi_2 : A\&B \to B$ by the obvious copycat strategies. Given $\sigma : C \to A$ and $\tau : C \to B$, define $\langle \sigma, \tau \rangle : C \to A\&B$ by

$$\begin{array}{rcl}\langle \sigma, \tau \rangle & = & \{s \in L_{C \multimap A\&B} \mid s \restriction C, A \in \sigma \wedge s \restriction B = \varepsilon\} \\ & \cup & \{s \in L_{C \multimap A\&B} \mid s \restriction C, B \in \tau \wedge s \restriction A = \varepsilon\}.\end{array}$$

Proposition 4 $A\&B$ is the product of A and B in each of the four categories $\mathcal{G}$, $\mathcal{G}_i$, $\mathcal{G}_b$ and $\mathcal{G}_{ib}$, with projections given by π_1 and π_2.

It should be clear how this definition generalizes to give all products.

3.3 Exponential

Definition Given a game A, define the game $!A$ as follows.

$$\begin{array}{rcl} M_{!A} & = & M_A \\ \lambda_{!A} & = & \lambda_A \\ \vdash_{!A} & = & \vdash_A \\ P_{!A} & = & \{s \in L_{!A} \mid \text{for each initial move } m, s \upharpoonright m \in P_A\}. \end{array}$$

3.3.1 Promotion

Given a map $\sigma : !A \to B$, we wish to define its promotion $\sigma^\dagger : !A \to !B$ to be a strategy which plays "several copies of σ". However, in general this cannot be done because there is no way for $\sigma^\dagger$ to know how the many threads of dialogue in $!A \multimap !B$ should be grouped together to give dialogues in $!A \multimap B$. There is a class of games B for which this can be done, however: the well-opened games.

Definition A game A is *well-opened* iff for all $sm \in P_A$ with m initial, $s = \varepsilon$.

In a well-opened game, initial moves can only happen at the first move, so there is only ever a single thread of dialogue. Note that if B is well-opened then so is $A \multimap B$ for any game A, so while $!A$ is not well-opened except in pathological cases, the game $!A \multimap B$ is well-opened whenever B is. We are going to construct a cartesian closed category in which all games are well-opened and exponentials (in the ordinary sense, not the linear logic one) are given by $!A \multimap B$, so this observation is important.

Given a map $\sigma : !A \to B$, define its promotion $\sigma^\dagger : !A \to !B$ by

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial } m, s \upharpoonright m \in \sigma\}.$$

Proposition 5 If A and B are well-opened games, and σ is a strategy for $!A \multimap B$, then $\sigma^\dagger$ is a strategy for $!A \multimap !B$. If σ is innocent then so is $\sigma^\dagger$; if σ is well-bracketed, so is $\sigma^\dagger$.

3.3.2 Dereliction

For well-opened games A, we can define $\mathsf{der}_A : !A \to A$ to be the copycat strategy

$$\{s \in P_{!A \multimap A} \mid \forall t \sqsubseteq^{\text{even}} s . t \upharpoonright !A = t \upharpoonright A\}.$$

Dereliction and promotion behave as expected where they are defined.

Proposition 6 Let A, B and C be well-opened games, and let $\sigma : !A \multimap B$ and $\tau : !B \multimap C$ be strategies. Then

- $\sigma^\dagger ; \mathsf{der}_B = \sigma$,
- $\mathsf{der}_A^\dagger = \mathsf{id}_{!A}$, and
- $\sigma^\dagger ; \tau^\dagger = (\sigma^\dagger ; \tau)^\dagger$.

We now note an important lemma.

Lemma 7 (Bang Lemma) If B is well-opened and $\sigma : !A \to !B$ is innocent then $\sigma = (\sigma ; \mathsf{der}_B)^\dagger$.

Proof First note that the set of P-views of positions of $!A \multimap !B$ is the same as that of $!A \multimap B$. It is straightforward to check that, when considered as a function from P-views to moves, any strategy $\sigma : !A \multimap !B$ is the same as $\sigma ; \mathsf{der}_B$, and a strategy $\tau : !A \multimap B$ is the same as $\tau^\dagger$. These two observations together prove the lemma. $\square$

3.3.3 Contraction

We define $\mathsf{con}_A : !A \to !A \otimes !A$. For any $t \in P_{!A_0 \multimap !A_1 \otimes !A_2}$, let I be the set of occurrences of initial moves in A_1 and J be the set of occurrences of initial moves in A_2. Let $t_1 = t \restriction I$ and $t_2 = t \restriction J$. Then define con_A as

$$\{s \in P_{!A_0 \multimap !A_1 \otimes !A_2} \mid \forall t \sqsubseteq^{\text{even}} s.(t_1 \restriction !A_0 = t_1 \restriction !A_1) \wedge (t_2 \restriction !A_0 = t_2 \restriction !A_2)\}.$$

3.3.4 Exponential isomorphisms

These reduce to *identities* in the present setting:

$$\begin{array}{rcl} !(A \& B) & = & !A \otimes !B. \\ I & = & !I. \end{array}$$

3.4 Four cartesian closed categories of games

We can now define the cartesian closed category of games $\mathcal{C}$, and its three subcategories $\mathcal{C}_i$, $\mathcal{C}_b$ and $\mathcal{C}_{ib}$. The category $\mathcal{C}$ is defined as follows.

Objects	:	Well-opened games
Morphisms $\sigma : A \to B$	:	Strategies for $!A \multimap B$

For any well-opened game A, the strategy $\mathsf{der}_A : !A \multimap A$ is the identity map on A, and given morphisms $\sigma : A \to B$ and $\tau : B \to C$, that is to say strategies $\sigma : !A \multimap B$ and $\tau : !B \multimap C$, we define the composite morphism $\sigma \fatsemi \tau : A \to C$ to be $\sigma^\dagger ; \tau$.

The subcategory $\mathcal{C}_i$ has as its morphisms only the innocent strategies; $\mathcal{C}_b$ has only the well-bracketed strategies; and $\mathcal{C}_{ib}$ has only those strategies which are both innocent and well-bracketed.

Products in all of these categories are constructed as in $\mathcal{G}$: set $A \times B = A \& B$. Moreover,

$$\begin{aligned} \mathcal{C}(A \times B, C) &= \mathcal{G}(!(A\&B), C) \\ &= \mathcal{G}(!A \otimes !B, C) \\ &\cong \mathcal{G}(!A, !B \multimap C) \\ &= \mathcal{C}(A, !B \multimap C). \end{aligned}$$

So we can define $A \Rightarrow B$ to be the game $!A \multimap B$, giving cartesian closure. Given a map $f : A \times B \to C$ we write $\Lambda(f) : A \to B \Rightarrow C$ for the map corresponding to f across this isomorphism, which we call *currying*.

Proposition 8 $\mathcal{C}$ is a cartesian closed category, with sub-cccs $\mathcal{C}_i$ and $\mathcal{C}_b$. The category $\mathcal{C}_{ib}$ is a sub-ccc of each of $\mathcal{C}$, $\mathcal{C}_i$ and $\mathcal{C}_b$.

The four cccs we have just defined of course form the four vertices of our "semantic square": $\mathcal{G}_{ib}$ corresponds to purely functional programming; $\mathcal{G}_i$ to functional programming with control operators; $\mathcal{G}_b$ to an extension of functional programming with local store; and $\mathcal{G}$ to programs with both store and control operators.

3.4.1 Order enrichment

The strategies for a game A are easily seen to form a directed-complete partial order under the inclusion ordering, with least element $\bot = \{\varepsilon\}$ and least upper bounds given by unions. Moreover, composition, tensor, currying etc. are all continuous with respect to this order. Applying this to the hom-objects $A \multimap B$, we obtain:

Proposition 9 $\mathcal{G}$ is a cpo-enriched autonomous category. $\mathcal{C}$ is a cpo-enriched cartesian closed category.

For any innocent strategy $\sigma : A$, define the *view-function* of σ to be the partial function f from P-views to P-moves defined by

$$f(v) = b \iff \exists sa.sab \in \sigma \wedge \ulcorner sa \urcorner = v.$$

(We are ignoring justification pointers here; strictly speaking the view function includes justification information.)

Proposition 10 The strategies for a game A form a dI-domain; the compact strategies are those with a finite set of positions. The same is true of the well-bracketed strategies.

The innocent strategies for a game A also form a dI-domain, with the compact strategies being those with finite view-function. The same is true of the innocent and well-bracketed strategies.

Note that an innocent strategy with finite view-function is not necessarily finite qua set of positions—this was the point of introducing view-functions in the first place. For example, by the Bang Lemma any innocent strategy for $!A$ other than $\bot$ has an infinite set of positions. Thus we shall speak of an innocent strategy $\sigma : A$ with finite view function being *innocently compact* i.e. compact in $\mathcal{G}_i(I, A)$ (and in $\mathcal{G}_{ib}(I, A)$ if it happens to be well-bracketed as well).

3.5 Intrinsic Preorder

Our full abstraction results will in fact hold not in $\mathcal{C}$ or its subcategories, but in quotients of these categories with respect to a certain preorder, which we now define. Let Σ be the game with a single question q and one answer a. There are only two strategies for Σ: $\bot = \{\varepsilon\}$ and $\top = \{\varepsilon, qa\}$, which are clearly both innocent and well-bracketed. Maps $\alpha : A \to \Sigma$ in the $\mathcal{C}$ or its subcategories can be thought of as tests on strategies for A: a strategy σ passes the test if $\sigma \mathbin{⨟} \alpha = \top$. The intrinsic preorder for strategies on A is defined as follows.

$$\sigma \lesssim \tau \text{ iff } \forall \alpha : A \to \Sigma . \sigma \mathbin{⨟} \alpha = \top \Rightarrow \tau \mathbin{⨟} \alpha = \top .$$

So $\sigma \lesssim \tau$ if τ passes every test passed by σ. Note that this defines four different preorders, one on each of $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}_b$ and $\mathcal{C}_{ib}$, because there is a different range of tests available in each of these categories. When necessary we will distinguish them using subscripts: for example, the preorder at work in $\mathcal{C}_{ib}$ will be written $\lesssim_{ib}$.

It is straightforward to show the following.

Proposition 11 Let $\mathbf{C}$ be one of $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}_b$ and $\mathcal{C}_{ib}$. The appropriate relation $\lesssim$ is a preorder on each hom-set of $\mathbf{C}$, and the quotient $\mathbf{C}/\lesssim$ of $\mathbf{C}$ by $\lesssim$ is a poset-enriched cartesian closed category.

4 The language PCF

The programming language **PCF** is a call-by-name functional language with a base type of expressions denoting natural numbers and constants for arithmetic and recursion. Its syntax is that of an applied simply-typed λ-calculus. Thus types, ranged over by A, B, ..., are given by the following grammar.

$$A, B, \ldots ::= \texttt{exp} \mid A \to B .$$

Terms, ranged over by M, N, ..., are as follows.

$$\begin{array}{lcl} M, N, \ldots & ::= & x \mid \lambda x : A.M \mid MN \\ & \mid & n \mid \texttt{succ}\ M \mid \texttt{pred}\ M \\ & \mid & \texttt{cond}\ M\ N_1\ N_2 \mid \texttt{Y}_A\ M \end{array}$$

Here x ranges over an infinite collection of variables, and n over the natural numbers. There are other variants of **PCF**: one could add a type of booleans and some operations on them, for example, but here we make do with the single base type `exp`.

Typing judgements take the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$$

where the variables appearing in the context $x_1 : A_1, \ldots, x_n : A_n$ are all distinct. Let Γ range over such contexts. The typing rules are as follows.

Variables

$$\frac{}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i}\; i \in \{1, \ldots, n\}$$

Functions

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Arithmetic

$$\frac{}{\Gamma \vdash n : \mathtt{exp}} \qquad \frac{\Gamma \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{succ}\ M : \mathtt{exp}} \qquad \frac{\Gamma \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{pred}\ M : \mathtt{exp}}$$

Conditional and recursion

$$\frac{\Gamma \vdash M : \mathtt{exp} \quad \Gamma \vdash N_1 : \mathtt{exp} \quad \Gamma \vdash N_2 : \mathtt{exp}}{\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2 : \mathtt{exp}} \qquad \frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathtt{Y}_A\ M : A}$$

As usual $\lambda x : A.M$ binds x in M, and we identify terms up to renaming of bound variables. We will often drop the type tags on $\lambda x : A.M$ and $\mathtt{Y}_A\ M$ when it will not cause confusion.

4.1 Operational semantics

The operational semantics of **PCF** is given in "big-step" style by means of a relation $M \Downarrow V$, where M ranges over closed terms (i.e. terms such that $\vdash M : A$ can be derived) and V ranges over *canonical forms*:

$$V ::= n \mid \lambda x.M.$$

Since **PCF** is deterministic, this relation determines a partial function from closed terms of type `exp`, that is, *programs* of **PCF**, to natural numbers. The relation is defined below. We write $M[N/x]$ for the capture-free substitution of term N for variable x in the term M.

Canonical forms

$$\frac{}{V \Downarrow V}$$

Functions

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

Arithmetic

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

Conditional

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

Recursion

$$\frac{M(\mathtt{Y}\ M) \Downarrow V}{\mathtt{Y}\ M \Downarrow V}$$

We write $M\Downarrow$ to indicate that $M \Downarrow V$ for some V. We can now define the *observational preorder* as follows. Given closed terms M and N of the same type, we write $M \precsim N$ iff for all contexts $C[-]$ (i.e. **PCF** terms with a hole $-$) such that $C[M]$ and $C[N]$ are well-formed closed terms of type `exp`, if $C[M]\Downarrow$ then $C[N]\Downarrow$. (We choose not to consider the observational preorder between open terms, although it can be extended to that case straightforwardly.)

4.2 Denotational semantics

Our categories of games are cartesian closed, so they have all the structure required to model the simply-typed λ-calculus. In this section we first review the interpretation of λ-calculus in a ccc and then go on to show how the constants of **PCF** can be accommodated in the categories of games.

A type A will be modelled as an object $[\![A]\!]$, and a well-typed open term $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ as a morphism

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

Given $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we write $[\![\Gamma]\!]$ for the product $[\![A_1]\!] \times \cdots \times [\![A_n]\!]$. In particular, the semantics of the empty context is the terminal object $\mathbf{1}$, so closed terms $M : A$ are modelled as maps $[\![M]\!] : \mathbf{1} \to [\![A]\!]$.

Once the interpretation $[\![\mathtt{exp}]\!]$ of the base type is fixed, the interpretation of other types is defined by induction: $[\![A \to B]\!] = [\![A]\!] \Rightarrow [\![B]\!]$.

Variables are interpreted using projections:

$$[\![x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i]\!] = \pi_i : [\![A_1]\!] \times \cdots \times [\![A_i]\!] \times \cdots \times [\![A_n]\!] \to [\![A_n]\!].$$

Abstraction is modelled by currying:

$$[\![\Gamma \vdash \lambda x : A.M : A \to B]\!] = \Lambda([\![\Gamma, x : A \vdash M : B]\!]) : [\![\Gamma]\!] \to [\![A]\!] \Rightarrow [\![B]\!].$$

The interpretation of application makes use of the *evaluation* map $\mathsf{ev} : (A \Rightarrow B) \times A \to B$ obtained by uncurrying the identity on $A \Rightarrow B$. If $\Gamma \vdash M : A \to B$ and $\Gamma \vdash N : A$ then

$$[\![\Gamma \vdash MN]\!] = \langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N]\!] \rangle \,;\, \mathsf{ev}.$$

We shall now show how **PCF** can be interpreted in any of the cartesian closed categories $\mathcal{C}, \mathcal{C}_i, \mathcal{C}_b$ and $\mathcal{C}_{ib}$. The interpretation of $\mathtt{exp}$ is the flat game $\mathbf{N}$ of natural numbers, defined as follows.

$$\begin{array}{rcll}
M_{\mathbf{N}} & = & \{q\} \cup \{n \mid n \in \omega\} & \\
\lambda_{\mathbf{N}}(q) & = & \mathsf{OQ} & \\
\lambda_{\mathbf{N}}(n) & = & \mathsf{PA} & \\
\star & \vdash_{\mathbf{N}} & q & \\
q & \vdash_{\mathbf{N}} & n & \text{for each } n \\
P_{\mathbf{N}} & = & \{\varepsilon, q\} \cup \{qn \mid n \in \omega\} &
\end{array}$$

Thus $\mathbf{N}$ has a single initial question q to which P can respond by playing a natural number. The strategies for $\mathbf{N}$ are $\bot = \{\varepsilon\}$ and $[\![n]\!] = \{\varepsilon, qn\}$ for each number n. These strategies are all innocent and well-bracketed, so we have an interpretation of the numeric constants in each of our cartesian closed categories.

The arithmetic operation $\mathtt{succ}$ is interpreted using the strategy depicted below.

$$\begin{array}{ccc}
!\mathbf{N} & \multimap & \mathbf{N} \\
 & & q \\
q & & \\
n & & \\
 & & n+1
\end{array}$$

This gives a map $s : [\![\mathtt{exp}]\!] \to [\![\mathtt{exp}]\!]$, so given $\Gamma \vdash M : \mathtt{exp}$, we set

$$[\![\Gamma \vdash \mathtt{succ}\ M : \mathtt{exp}]\!] = [\![\Gamma \vdash M]\!] \,;\, s : [\![\Gamma]\!] \to [\![\mathtt{exp}]\!].$$

The operation `pred` is handled similarly. As for the conditional, we use the strategy whose two typical plays are shown below.

$$
\begin{array}{ccccccc}
!\mathbf{N} & \otimes & !\mathbf{N} & \otimes & !\mathbf{N} & \multimap & \mathbf{N} \\
 & & & & & & q \\
q & & & & & & \\
0 & & & & & & \\
 & & q & & & & \\
 & & n & & & & \\
 & & & & & & n \\
\hline
 & & & & & & q \\
q & & & & & & \\
m \neq 0 & & & & & & \\
 & & & & q & & \\
 & & & & n & & \\
 & & & & & & n
\end{array}
$$

This strategy responds to the initial question by asking the question in the first $!\mathbf{N}$. If O plays 0, the second $!\mathbf{N}$ is interrogated, and any response it copied to the output; if O responds with a non-zero number m in the first $!\mathbf{N}$, the third $!\mathbf{N}$ is interrogated instead, and again any response is copied to the output. This strategy defines a map $c : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$, since $!(\mathbf{N} \times \mathbf{N} \times \mathbf{N}) = !\mathbf{N} \otimes !\mathbf{N} \otimes !\mathbf{N}$. We can then interpret $\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2$ as

$$[\![\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2]\!] = \langle [\![\Gamma \vdash M]\!], [\![\Gamma \vdash N_1]\!], [\![\Gamma \vdash N_2]\!] \rangle \fatsemi c.$$

To interpret recursively defined terms $\mathtt{Y}\ M$ we use the fact that our categories are cpo-enriched. The interpretation of a term $\Gamma \vdash M : A \to A$ determines a map $f : [\![\Gamma]\!] \times [\![A]\!] \to [\![A]\!]$ by uncurrying. We now define a chain of maps $f_n : [\![\Gamma]\!] \to [\![A]\!]$ as follows:

$$
\begin{array}{rcl}
f_0 & = & \bot : [\![\Gamma]\!] \to [\![A]\!] \\
f_{n+1} & = & \langle \mathsf{id}_{[\![\Gamma]\!]}, f_n \rangle \fatsemi f.
\end{array}
$$

and set $[\![\mathtt{Y}\ M]\!]$ to be the least upper bound of this chain, that is, the least fixed point of the operation taking a map $g : [\![\Gamma]\!] \to [\![A]\!]$ to $\langle \mathsf{id}_{[\![\Gamma]\!]}, g \rangle \fatsemi f$. Note in particular that $[\![\mathtt{Y}_A\ (\lambda x : A.x)]\!] = \bot$. We shall write Ω_A for the term $\mathtt{Y}_A\ (\lambda x : A.x)$.

This completes our interpretation of **PCF**. The reader should check that all the strategies used are well-bracketed and innocent, so the interpretation is valid in any of our cccs.

4.3 Soundness and Adequacy

The aim of this section is to prove that for any closed terms M and N of type A,

$$[\![M]\!] \subseteq [\![N]\!] \Rightarrow M \sqsubseteq\!\!\!\!\sim N.$$

Thus our models of **PCF** can be used to reason about the observational preorder.

The first step is a substitution lemma.

Lemma 12 If $\Gamma, x : A \vdash M : B$ and $\Gamma \vdash N : A$ are well-typed terms, then so is $\Gamma \vdash M[N/x] : B$ and

$$[\![\Gamma \vdash M[N/x]]\!] = \langle \mathsf{id}_{[\![\Gamma]\!]}, [\![\Gamma \vdash N]\!] \rangle \fatsemi [\![\Gamma, x : A \vdash M]\!].$$

Proof A straightforward induction, using naturality of currying and continuity of composition for the second part. $\square$

We can now show that our denotational semantics respects the operational semantics, that is, that the model is *sound* for evaluation.

Lemma 13 If $M \Downarrow V$ then $[\![M]\!] = [\![V]\!]$.

Proof Another induction, this time on the derivation of $M \Downarrow V$. We shall treat the case of application. The rule in question is shown below.

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

By the inductive hypothesis, $[\![M]\!] = [\![\lambda x.M']\!] = \Lambda([\![x \vdash M]\!])$, and $[\![M'[N/x]]\!] = [\![V]\!]$. We now calculate as follows.

$$\begin{array}{rcll}
[\![MN]\!] & = & \langle [\![M]\!], [\![N]\!] \rangle \fatsemi \mathsf{ev} & \\
& = & \langle \Lambda([\![x \vdash M']\!]), [\![N]\!] \rangle \fatsemi \mathsf{ev} & \\
& = & \langle \mathsf{id}, [\![N]\!] \rangle \fatsemi (\Lambda([\![x \vdash M']\!]) \times \mathsf{id}) \fatsemi \mathsf{ev} & \\
& = & \langle \mathsf{id}, [\![N]\!] \rangle \fatsemi [\![x \vdash M']\!] & \\
& = & [\![M'[N/x]]\!] & \text{by Lemma 12} \\
& = & [\![V]\!] & \square
\end{array}$$

So for programs $\vdash M : \mathtt{exp}$, we have $M\Downarrow \Rightarrow [\![M]\!] \neq \perp$. We now show that the converse of this also holds: if $[\![M]\!] \neq \perp$ then $M\Downarrow$. We call such results *computational adequacy*. The proof uses Plotkin's method of a *computability predicate* [33] on terms, which we now define.

Definition

- A closed term $\vdash M : \mathtt{exp}$ is computable if whenever $[\![M]\!] \neq \perp$ it is the case that $M\Downarrow$.
- A closed term $\vdash M : A \to B$ is computable if $\vdash MN : B$ is computable for all computable $\vdash N : A$.
- An open term $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ is computable if for all closed computable $N_1 : A_1, \ldots, N_n : A_n$, the term $\vdash M[N_1/x_1, \ldots, N_n/x_n] : A$ is computable.

Our goal is to show that all terms of **PCF** are computable. We shall first do this for terms of a restricted language in which the only allowed use of the $\mathtt{Y}$ combinator is in terms of the form $\mathtt{Y}\ (\lambda x.x)$; that is, the terms Ω_A are included in this sublanguage, but no other use of $\mathtt{Y}$ is. We call this restricted language $\mathbf{PCF}_1$.

Lemma 14 All terms of $\mathbf{PCF}_1$ are computable.

Proof A straightforward induction, using the fact that $[\![\Omega]\!] = \bot$ and that if $[\![M : A \to B]\!] = \bot$ then $[\![MN]\!] = \bot$ for any $N : A$. $\square$

We now lift this result to full **PCF**. Given a term $\Gamma \vdash M : A \to A$ of **PCF**, we define a sequence of *syntactic approximants* to $\mathtt{Y}_A\ M$ by

$$\begin{array}{rcl} \mathtt{Y}^0_A\ M & = & \Gamma \vdash \Omega_A : A \\ \mathtt{Y}^{n+1}_A\ M & = & M(\mathtt{Y}^n_A\ M). \end{array}$$

We now define the relation of syntactic approximation between terms, $M \prec N$, as follows.

$$\frac{}{M \prec M} \qquad \frac{M \prec N}{\mathtt{Y}^n\ M \prec \mathtt{Y}\ N}$$

plus rules expressing congruence of $\prec$ with respect to the term forming operations of **PCF**. For example,

$$\frac{M \prec M' \quad N_1 \prec N_1' \quad N_2 \prec N_2'}{\mathtt{cond}\ M\ N_1\ N_2 \prec \mathtt{cond}\ M'\ N_1'\ N_2'.}$$

Lemma 15 If $M \prec M'$ and $M \Downarrow V$ then $M' \Downarrow V'$ for some V' with $V \prec V'$.

Proof Induction on the derivation of $M \Downarrow V$. $\square$

For any **PCF** term M and natural number n, define M_n to be the $\mathbf{PCF}_1$ term obtained from M by replacing each subterm of the form $\mathtt{Y}\ N$ with $\mathtt{Y}^n\ N_n$. Note that $M_n \prec M$. The compositionality and continuity of our semantics now gives:

Lemma 16 For any term M, $[\![M]\!] = \bigcup_{n\in\omega}[\![M_n]\!]$.

We can finally prove that all **PCF** terms are computable.

Lemma 17 All terms of **PCF** are computable.

Proof It suffices to show that all closed terms of type $\mathtt{exp}$ are computable. Let $\vdash M : \mathtt{exp}$ be such a term. By Lemma 16, $[\![M]\!] = \bigcup[\![M_n]\!]$ where each M_n is computable by Lemma 14. If $[\![M]\!] \neq \bot$ then for some n, $[\![M_n]\!] \neq \bot$, and therefore $M_n\Downarrow$. But $M_n \prec M$, so $M\Downarrow$ by Lemma 15. Hence M is computable. $\square$

Soundness and adequacy together imply the *inequational soundness* result which is our aim.

Proposition 18 If $\vdash M : A, \vdash N : A$ and $[\![M]\!] \subseteq [\![N]\!]$ then $M \precsim N$.

Proof Suppose $[\![M]\!] \subseteq [\![N]\!]$ and $C[M]\Downarrow$ for some context $C[-]$. Then by the compositionality of our semantics, $[\![C[M]]\!] \subseteq [\![C[N]]\!]$ and $[\![C[M]]\!] \neq \bot$ by soundness, so $[\![C[N]]\!] \neq \bot$. But $C[N]$ is computable, so $C[N]\Downarrow$. Hence $M \precsim N$. $\square$

4.4 Definability

Up to now our results have held equally for models of **PCF** in $\mathcal{C}$, $\mathcal{C}_i$, $\mathcal{C}_b$ and $\mathcal{C}_{ib}$. We now concentrate on $\mathcal{C}_{ib}$, with the aim of showing that

> every compact element of the model in $\mathcal{C}_{ib}$ is of the form $[\![M]\!]$ for some term M.

We shall not give all the details here, but will highlight the important points of the proof. The interested reader should consult [1,14,26,27] for a fuller account, and [2] for an axiomatization of the essential properties of $\mathcal{C}_{ib}$ and $\mathcal{G}_{ib}$ which are used.

In fact the claim above is not quite true for the version of **PCF** we have presented, because of its economical syntax. For example, there is no term of **PCF** which denotes a ternary conditional strategy with typical plays as shown below.

$$\begin{array}{ccccccccc}
!\mathbf{N} & \otimes & !\mathbf{N} & \otimes & !\mathbf{N} & \otimes & !\mathbf{N} & \multimap & \mathbf{N}\\
&&&&&&&& q\\
q &&&&&&&&\\
0 &&&&&&&&\\
&& q &&&&&&\\
&& n &&&&&&\\
&&&&&&&& n\\
\hline
&&&&&&&& q\\
q &&&&&&&&\\
1 &&&&&&&&\\
&&&& q &&&&\\
&&&& n &&&&\\
&&&&&&&& n\\
\hline
&&&&&&&& q\\
q &&&&&&&&\\
m>1 &&&&&&&&\\
&&&&&& q &&\\
&&&&&& n &&\\
&&&&&&&& n
\end{array}$$

To remedy this, we add to **PCF** k-ary conditionals

$$\mathtt{case}_k\ M\ N_1\ N_2 \ldots N_k$$

with operational semantics as follows.

$$\frac{M \Downarrow i \quad N_{i+1} \Downarrow V}{\mathtt{case}_k\ M\ N_1 \ldots N_k \Downarrow V}\ i \in \{0, \ldots, k-1\}$$

The denotation of such a term is given by the strategy outlined above. This is a harmless extension of **PCF** because it is *conservative* with respect to the observational preorder: if M and N are terms such that for all $\mathtt{case}$-free contexts $C[-]$, $C[M]\Downarrow \Rightarrow C[N]\Downarrow$, then the same is true for *all* contexts. This holds because a term

$$M_1 = \mathtt{case}_k M\ N_1 \ldots N_k$$

can be replaced by

$$M_2 = \texttt{cond}\ M\ N_1(\texttt{cond}(\texttt{pred}\ M)\ N_2\ (\texttt{cond}\ldots(\texttt{cond}\ (\texttt{pred}^{k-1}\ M)\ N_k\ \Omega)))$$

which has the same operational semantics. We call the extended language **PCF**′.

Exercise

1. Prove the above claim.
2. Prove that $[\![M_1]\!] \neq [\![M_2]\!]$ in the case $k = 3$.

In what follows we shall use the linear type structure of $\mathcal{G}_{ib}$ to analyse the model of **PCF** in $\mathcal{C}_{ib}$, so most of the time we are working in $\mathcal{G}_{ib}$. We shall also frequently identify a **PCF** type A with the game $[\![A]\!]$ which it denotes.

Proposition 19 Let $A_1, \ldots, A_n$ be **PCF** types and $\sigma : !A_1 \otimes \cdots \otimes !A_i \multimap (!A_{i+1} \multimap \cdots \multimap !A_n \multimap \mathbf{N})$ an innocent, well-bracketed strategy with finite view-function. There exists a term

$$x_1 : A_1, \ldots, x_i : A_i \vdash M : A_{i+1} \to \ldots \to A_n \to \texttt{exp}$$

of **PCF**′ such that $[\![M]\!] = \sigma$.

Proof By induction on the size of the view-function of σ. In the base case, $\sigma = \perp$ so we can set $M = \Omega$. For the inductive step, there are two possible cases. Either σ responds to the initial question with some answer k, in which case $\sigma = [\![x_1, \ldots, x_i \vdash \lambda x_{i+1} : A_{i+1} \ldots \lambda x_n : A_n.k]\!]$, or σ replies to the initial question q by asking the question q' in some $!A_j$. We shall now analyse this case in detail. The idea is that σ corresponds to a `case` statement, first investigating the argument which O supplies in the A_i component, and then branching according to the response.

First let us uncurry σ, arriving at a strategy

$$\sigma' : !A_1 \otimes \cdots \otimes !A_n \multimap \mathbf{N}.$$

We shall find a term $x_1 : A_1, \ldots, x_N : A_n \vdash M : \texttt{exp}$ such that $\sigma' = [\![M]\!]$, and the required result follows by λ-abstracting M. The strategy σ responds to the initial q with q' in $!A_j$. In the play which follows, some moves of $!A_j$ will be justified by this occurrence of q'. We can separate out all such moves into an extra copy of A_j, simply by relabelling the moves in question, arriving at a strategy

$$\sigma'' : !A_1 \otimes \cdots \otimes !A_n \otimes A_j \multimap \mathbf{N}$$

which responds to the initial question q with q' in this new copy of A_j. Note the linear type of the new A_j: only one initial move q' is ever played there.

Now consider the odd-length sequences of moves of this game when P plays according to σ''. Such a sequence begins qq', and the P-view of the sequence either contains an immediate answer to q', or no answer to q' at all. In the former case, the views take the form $qq'ks$, and give rise to views qs of positions in the game

$$!A_1 \otimes \cdots \otimes !A_n \multimap \mathbf{N}.$$

(We have simply deleted all play in the extra copy of the game A_j.) Thus, for each k, the responses of σ'' at views $qq'ks$ give rise to an innocent strategy τ_k for the game above. These τ_k are the strategies corresponding to the branches of the case statement which will represent σ. The view function of each τ_k is smaller than that of σ, so by the inductive hypothesis, $\tau_k = [\![N_k]\!]$ for some term N_k. Furthermore, since the view function of σ is finite, there is some L such that for any $k \geq L$, $\tau_k = \bot$.

Now consider the views $qq's$ in which q' is not answered. Since σ is well-bracketed, q is not answered either. Therefore the sequence s consists entirely of moves in $!A_1, \ldots, !A_n$ and the separate A_j. If $A_j = B_1 \to \cdots \to B_l \to \texttt{exp}$, these moves must in fact be in the $!B_m$ components. Relabelling moves, the sequence s can be construed as a play in the game

$$!A_1 \otimes \cdots \otimes !A_n \multimap !B_1 \otimes \cdots \otimes !B_l$$

and the set of responses made by σ'' at such views gives rise to an innocent strategy for this game. But $!B_1 \otimes \cdots \otimes !B_l = !(B_1 \times \cdots \times B_l)$, so by Lemma 7 and the universal property of products, this strategy is the promotion of a strategy

$$\langle \alpha_1, \ldots, \alpha_l \rangle : !A_1 \otimes \cdots \otimes !A_n \multimap B_1 \times \cdots \times B_l.$$

Each α_r is innocent and well-bracketed and has smaller view-function than σ, so $\alpha_r = [\![M_r]\!]$ for some term M_r by the inductive hypothesis. It can then be checked that

$$\sigma' = [\![\texttt{case}_L\ (x_j M_1 \ldots M_l)\ N_0\ N_1 \ldots N_{L-1}]\!]. \qquad \square$$

4.5 Full Abstraction

We can now show that the category $\mathcal{C}_{ib}/\lesssim_{ib}$ contains a fully abstract model of **PCF**. In fact, there is some work to be done to show that $\mathcal{C}_{ib}/\lesssim_{ib}$ is a model of **PCF** at all. The reason is that although the cartesian closed structure and interpretations of the base type, arithmetic constants and conditional of **PCF** all survive the quotient from $\mathcal{C}_{ib}$ to $\mathcal{C}_{ib}/\lesssim_{ib}$, it is not known whether the quotiented category is cpo-enriched. We must therefore take care in interpreting recursion. Given a term $\Gamma \vdash M : A \to A$, we can form the chain of maps $f'_n : \Gamma \to A$ in $\mathcal{C}_{ib}/\lesssim_{ib}$ in just the same way as the chain f_n is formed in $\mathcal{C}_{ib}$. It is easy to show that $f'_n = [f_n]$, the equivalence class of f_n, and hence that the chain f'_n has a least upper bound given by $[\bigcup f_n]$. A category in which such chains have least upper bounds which are preserved by composition is called *rational*; for more information on rational categories, see [1, 27].

Rather than dwelling on the issue of how the quotient gives rise to a model, we shall go ahead and prove the following result.

Theorem 20 For any closed **PCF** terms M and N of the same type,

$$[\![M]\!] \lesssim_{ib} [\![N]\!] \iff M \sqsubseteq N.$$

Proof First notice that in defining the intrinsic preorder, we could let the object of tests be $\mathbf{N}$ rather than Σ, and say that σ passes test α if $\sigma \mathbin{;} \alpha \neq \bot$, with the same effect. The left-to-right direction of the theorem is now straightforward. Suppose $[\![M]\!] \lesssim_{ib} [\![N]\!]$ and that $C[M]\Downarrow$ for some context $C[-]$. The term $C[x]$ (where x is fresh) determines a map $\alpha : A \to \mathbf{N}$ such that $[\![C[P]]\!] = [\![P]\!] \mathbin{;} \alpha$ for all suitably typed closed terms P. We have $[\![M]\!] \mathbin{;} \alpha \neq \bot$, so since $[\![M]\!] \lesssim_{ib} [\![N]\!]$, we also have $[\![N]\!] \mathbin{;} \alpha \neq \bot$. But $C[N]$ is computable, so $C[N]\Downarrow$ as required.

For the right-to-left direction, we prove the contrapositive. Suppose $[\![M]\!] \not\lesssim_{ib} [\![N]\!]$. Then for some $\alpha : A \to \mathbf{N}$, $[\![M]\!] \mathbin{;} \alpha \neq \bot$ and $[\![N]\!] \mathbin{;} \alpha = \bot$. The strategy α can be taken to be compact, and then by Proposition 19, $\alpha = [\![x : A \vdash C[x]]\!]$ for some term $C[x]$. We therefore have $[\![C[M]]\!] \neq \bot$ and $[\![C[N]]\!] = \bot$, so $C[M]\Downarrow$ but not $C[N]\Downarrow$, as required. $\square$

5 Idealized Algol

We now consider a simplified version of the language introduced by Reynolds in [34], variants of which have come to be known as Idealized Algol (**IA** for short). Idealized Algol is a block-structured imperative programming language with higher-order procedures; otherwise put, it is an extension of **PCF** with the constructs of a basic imperative language and block-allocated local variables. Our game semantics of **IA** will therefore be an extension of that of **PCF** with suitable types and constants. The main result of this section is that the category $\mathcal{C}_b$ of well-bracketed but not necessarily innocent strategies gives rise (after taking the quotient with respect to the intrinsic preorder) to a fully abstract model of our variant of **IA**. Thus we see that the addition of imperative constructs to **PCF** and the relaxation of innocence are strongly related.

Idealized Algol is obtained by adding to **PCF** two base types: the type `com`, for *commands* which alter the state, and `var`, for *variables* which store natural numbers. We also add a constant `skip` : `com`, the "do nothing" command, and an operation for sequential composition:

$$\frac{\Gamma \vdash M : \mathtt{com} \quad \Gamma \vdash N : A}{\Gamma \vdash \mathtt{seq}_A\ M\ N : A}\ A = \mathtt{com} \text{ or } A = \mathtt{exp}$$

Intuitively, `seq` $M\ N$ runs the command M until it terminates and then behaves like N. Thus $\mathtt{seq}_{\mathtt{com}}\ C_1\ C_2$ is the usual sequential composition of commands, while $\mathtt{seq}_{\mathtt{exp}}$ allows expressions to have side-effects. This is perhaps the most important difference between our language and that originally proposed by Reynolds, in which a sharp distinction is drawn between commands, which affect the store but return no interesting value, and expressions, which may not alter the store. It is possible to give a fully abstract games model for the language without $\mathtt{seq}_{\mathtt{exp}}$, but this requires further sophistication in the model, which we do not wish to consider here; see [7].

We also have operations for writing to and reading from the store.

$$\frac{\Gamma \vdash M : \mathtt{var} \quad \Gamma \vdash N : \mathtt{exp}}{\Gamma \vdash \mathtt{assign}\ M\ N : \mathtt{com}} \qquad \frac{\Gamma \vdash M : \mathtt{var}}{\Gamma \vdash \mathtt{deref}\ M : \mathtt{exp}}$$

Variables are allocated using a binding operator $\mathtt{new}\ x\ \mathtt{in}\ M$:

$$\frac{\Gamma, x : \mathtt{var} \vdash M : A}{\Gamma \vdash \mathtt{new}\ x\ \mathtt{in}\ M : A}\ A = \mathtt{com} \text{ or } A = \mathtt{exp}$$

In such a term, the variable x is bound in M.

The final constructor we add to our language requires some explanation. The work of Reynolds and others (cf. [32]) takes the view that a variable is an "object" with two methods: the write method, which takes a number as input and executes a command to store that number, and the read method, which returns the number stored in the variable. We therefore have

$$\mathtt{var} = \mathtt{acc} \times \mathtt{exp}$$

where $\mathtt{acc}$ is a type of *acceptors* or write-methods. An acceptor can be seen as a (strict) function from expressions to commands: the acceptor for a variable x takes a number n and executes the command to store n in x. To make this view of variables explicit in the language, we add a constructor

$$\frac{\Gamma \vdash M_1 : \mathtt{exp} \to \mathtt{com} \quad \Gamma \vdash M_2 : \mathtt{exp}}{\Gamma \vdash \mathtt{mkvar}\ M_1\ M_2 : \mathtt{var}.}$$

Unlike the addition of the $\mathtt{case}_k$ constructs to **PCF** in the previous section, adding $\mathtt{mkvar}$ to the language *does* alter the observational preorder; the problem of finding a fully abstract model for the language without $\mathtt{mkvar}$ remains open.

5.1 Operational semantics

The operational semantics of **IA** is given in terms of *stores*. Given a context $\Gamma = x_1 : \mathtt{var}, \ldots, x_n : \mathtt{var}$ in which all variables have type $\mathtt{var}$ (a $\mathtt{var}$-*context*), a Γ-*store* s is a function from $\{x_1, \ldots, x_n\}$ to natural numbers. We write $(s \mid x \mapsto n)$ for the store identical to s but that x is mapped to n; this operation is used both to update a Γ-store and to extend a Γ-store to a Γ, x-store.

The only canonical form of type $\mathtt{com}$ is $\mathtt{skip}$, while the canonical forms of type $\mathtt{var}$ are variables and all terms of the form $\mathtt{mkvar}\ M\ N$. We now define a relation

$$s, M \Downarrow s', V$$

where $\Gamma \vdash M : A$ is a term and $\Gamma \vdash V : A$ a canonical form, with Γ a $\mathtt{var}$-context and s and s' both Γ-stores. Using the convention that a rule such as

$$\frac{M_1 \Downarrow V_1 \quad M_2 \Downarrow V_2}{M \Downarrow V}$$

is an abbreviation for

$$\frac{s, M_1 \Downarrow s', V_1 \quad s', M_2 \Downarrow s'', V_2}{s, M \Downarrow s'', V}$$

the rules for Idealized Algol are those for **PCF** together with the following.

Sequencing

$$\frac{M \Downarrow \texttt{skip} \quad N \Downarrow V}{\texttt{seq}\ M\ N \Downarrow V}$$

Variables

$$\frac{s, N \Downarrow s', n \quad s', M \Downarrow s'', x}{s, \texttt{assign}\ M\ N \Downarrow (s'' \mid x \mapsto n), \texttt{skip}} \qquad \frac{s, M \Downarrow s', x}{s, \texttt{deref}\ M \Downarrow s', n}\ s'(x) = n$$

`mkvar`

$$\frac{N \Downarrow n \quad M \Downarrow \texttt{mkvar}\ M_1\ M_2 \quad M_1 n \Downarrow \texttt{skip}}{\texttt{assign}\ M\ N \Downarrow \texttt{skip}} \qquad \frac{M \Downarrow \texttt{mkvar}\ M_1\ M_2 \quad M_2 \Downarrow n}{\texttt{deref}\ M \Downarrow n}$$

Blocks

$$\frac{(s \mid x \mapsto 0), M \Downarrow (s' \mid x \mapsto n), V}{s, \texttt{new}\ x\ \texttt{in}\ M \Downarrow s', V}$$

Notice that the rule for assignment evaluates the expression part before the variable part. This choice has been made to fit with the game semantics to follow. It is not clear how to adapt the model to the other choice of operational semantics: the problem of finding a fully abstract model for the other evaluation order remains open!

For a closed term M, we write $M\Downarrow$ if $M \Downarrow V$ in the unique (empty) store, and define the observational preorder just as for **PCF**, but using closed contexts of type `com` rather than `exp`. Notice that this choice of type makes no difference for this language in which expressions may have side-effects; but in the restricted language without the $\texttt{seq}_{\texttt{exp}}$ construct, this decision is vital.

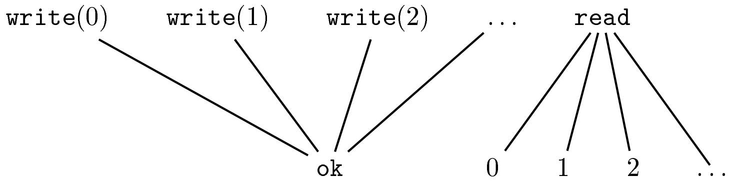
5.2 Denotational semantics

We now show how to extend the semantics of **PCF** to obtain models of **IA** in $\mathcal{C}_b$ and $\mathcal{C}$. The extension requires strategies which are not innocent, so the categories $\mathcal{C}_i$ and $\mathcal{C}_{ib}$ cannot be used.

The type `com` is interpreted as a two-move game, with a single initial move `run` enabling a single answer `done`: aside from effects on the store, the only observation to be made of a command is its termination. For the type `var`, we appeal to Reynolds's analysis

$$\mathtt{var} = \mathtt{acc} \times \mathtt{exp}.$$

It therefore remains for us to give an interpretation of the type of acceptors. As indicated earlier, an acceptor can be seen as a strict function from expressions to commands. Such a function can alternatively be presented as a collection of commands, one for each natural number. Thus we model `acc` as $\mathtt{com}^{\omega}$, the product of countably many `com` games. Renaming the moves to make them more suggestive, the game for `var` has an initial move $\mathtt{write}(n)$ for each natural number n, and an initial move `read`. Each $\mathtt{write}(n)$ enables a single answer `ok`, while `read` enables an answer n for each natural number. In a picture:



Notice the elegant duality between the read and write parts of this game. [This interpretation of the type of acceptors is slightly different from that used in our informal introduction, which was simply $\mathbf{N} \Rightarrow \mathtt{com}$. Using this type would allow *non-strict* acceptors, which are not present in **IA**, so full abstraction would fail.]

The interpretation of `skip` is the strategy which responds to `run` with `done` immediately. Sequential composition is interpreted by the obvious strategy: in the case of $\mathtt{seq}_{\mathtt{exp}}$, it is as shown below.

$$\begin{array}{ccccc}
!\mathtt{com} & \otimes & !\mathtt{exp} & \multimap & \mathtt{exp} \\
 & & & & q \\
\mathtt{run} & & & & \\
\mathtt{done} & & & & \\
 & & q & & \\
 & & n & & \\
 & & & & n
\end{array}$$

Given our interpretation of `var`, the denotations of assignment and dereferencing should be clear:

$$\begin{array}{ccccc}
 & & & \mathtt{assign} & \\
!\mathtt{var} & \otimes & !\mathtt{exp} & \multimap & \mathtt{com} \\
 & & & & \mathtt{run} \\
 & & \mathtt{read} & & \\
 & & n & & \\
\mathtt{write}(n) & & & & \\
\mathtt{ok} & & & & \\
 & & & & \mathtt{done}
\end{array}
\qquad
\begin{array}{ccc}
 & \mathtt{deref} & \\
!\mathtt{var} & \multimap & \mathtt{exp} \\
 & & q \\
\mathtt{read} & & \\
n & & \\
 & & n
\end{array}$$

Notice that the strategy for assignment moves in $!\mathtt{exp}$ before $!\mathtt{var}$, which is why the expression part must be evaluated first in the corresponding operational semantics. For `mkvar`, since $\mathtt{var} = \mathtt{acc} \times \mathtt{exp}$, we use pairing together with the following coercion from $!\mathtt{exp} \multimap \mathtt{com}$ to $\mathtt{com}^{\omega}$:

$$
\begin{array}{ccccc}
(!\mathtt{exp} & \multimap & \mathtt{com}) & \longrightarrow & \mathtt{com}^{\omega} \\
 & & & & \mathtt{write}(n) \\
 & & \mathtt{run} & & \\
q & & & & \\
n & & & & \\
 & & \mathtt{done} & & \\
 & & & & \mathtt{ok}
\end{array}
$$

All of the above strategies are innocent, and none of them takes any account of state: they all just copy and manipulate data without "remembering" anything. All of the work in modelling state is in fact done by the interpretation of `new`, which is after all the only source of stateful behaviour in the language.

5.2.1 Interpreting `new`

The denotation of `new` in the games model is built from a "storage cell" strategy `cell`. First thoughts suggest that `cell` should be a strategy for `var`, since it is supposed to be the interpretation of a locally allocated variable. The key aspect of storage cells, however, is the way their contents persist from one use to the next, so we in fact use the game $!\mathtt{var}$, which can be thought of as a type of "reusable `var`s".

Plays in the game $!\mathtt{var}$ take the form:

$$\mathtt{read} \cdot 3 \cdot \mathtt{write}(2) \cdot \mathtt{ok} \cdot \mathtt{read} \cdot 9 \cdot \mathtt{read} \cdot 7 \cdot \mathtt{write}(1) \cdot \mathtt{ok} \cdots$$

with no causal relationship between the various values written and read. The strategy `cell` just imposes the obvious storage cell behaviour: it responds to each move of the form $\mathtt{write}(-)$ with `ok`, and responds to a `read` with the last value written, if any; if no value has yet been written, `cell` responds to `read` with 0, since our operational semantics initializes variables to zero. Other choices of operational semantics could be reflected by different initializations of this `cell` strategy. For emphasis, we shall sometimes write $\mathtt{cell}_0$ for this version of `cell`, and $\mathtt{cell}_n$ for the version initialized to n.

The strategy $\mathtt{cell} : I \multimap\ !\mathtt{var}$ is well-defined and well-bracketed, but it is *not* innocent, because after each `read` move, the P-view consists just of that `read`, so an innocent strategy could not hope to return the last value written into the cell.

Given a term $\Gamma, x : \mathtt{var} \vdash M : \mathtt{com}$ (or `exp`), with interpretation $m : !\Gamma \otimes !\mathtt{var} \to \mathtt{com}$ we can interpret $\mathtt{new}\ x\ \mathtt{in}\ M$ by

$$!\Gamma \xrightarrow{\ \cong\ } !\Gamma \otimes I \xrightarrow{\ \mathsf{id} \otimes \mathtt{cell}\ } !\Gamma \otimes !\mathtt{var} \xrightarrow{\ m\ } \mathtt{com}$$

Remark Although we have made use of the linear category $\mathcal{G}$ in defining our model of **IA**, it should be emphasised that the model is in the cartesian closed category $\mathcal{C}$ (and in $\mathcal{C}_b$). The underlying linear type structure is useful in analysing and discussing the model, because it gives direct access to the map $\mathtt{cell} : I \multimap !\mathtt{var}$. It would be possible to do without this extra expressivity, defining a map $\mathtt{new} : \mathtt{var} \Rightarrow \mathtt{com} \to \mathtt{com}$ in $\mathcal{C}$ directly. We have chosen not to do so because the $\mathtt{cell}$ map is of great use in the proof of soundness which follows.

5.3 Soundness and Adequacy

As for **PCF**, the first step on the way to a soundness result is to show that the model respects the relation $s, M \Downarrow s', V$ of the operational semantics. However, there is a mismatch between the elements of this relation, which involve explicit stores s and s', and the *implicit* way in which state is represented in the games model using history-sensitivity of strategies. To bridge this gap, we massage the game semantics a little to make states more explicit.

Given a $\mathtt{var}$-context $\Gamma = x_1 : \mathtt{var}, \ldots, x_n : \mathtt{var}$ and a Γ-store s, we define the interpretation $[\![s]\!]$ of s to be a map

$$I \xrightarrow{[\![s]\!]} \overbrace{!\mathtt{var} \otimes \cdots \otimes !\mathtt{var}}^{n}$$

of $\mathcal{G}_b$ (or $\mathcal{G}$) consisting of a tuple of suitably initialized $\mathtt{cell}$ strategies:

$$I \xrightarrow{\cong} I \otimes \cdots \otimes I \xrightarrow{\mathtt{cell}_{s(0)} \otimes \cdots \otimes \mathtt{cell}_{s(n)}} !\mathtt{var} \otimes \cdots \otimes !\mathtt{var}.$$

We can now give an interpretation to a configuration s, M where $\Gamma \vdash M : A$ is a term and s is a Γ-store, using the composition

$$I \xrightarrow{[\![s]\!]} !\Gamma \xrightarrow{[\![M]\!]} A.$$

Our soundness result will be that whenever $s, M \Downarrow s', V$ we have $[\![s]\!] ; [\![M]\!] = [\![s']\!] ; [\![V]\!]$. However, the proof of this fact requires a slightly stronger inductive hypothesis. For example if V is some constant which does not depend on the store at all, then $[\![s']\!] ; [\![V]\!] = [\![s'']\!] ; [\![V]\!]$ for all stores s''! If M is used in a context which makes use of the store after evaluating M, the fact that s, M evaluates to s', V rather than s'', V will be crucial, so we must reflect this in our inductive hypothesis.

Consider instead the strategy $[\![s, M]\!]$ defined to be the map below.

$$I \xrightarrow{[\![s]\!]} !\Gamma \xrightarrow{\mathsf{con}} !\Gamma \otimes !\Gamma \xrightarrow{[\![M]\!] \otimes \mathsf{id}} A \otimes !\Gamma.$$

The extra copy of $!\Gamma$ provided by the contraction map serves to allow access to the store from outside. We can now prove the following soundness result.

Lemma 21 Given a term $\Gamma \vdash M : A$ and a Γ-store s such that $s, M \Downarrow s', V$, the plays of $[\![s, M]\!] : I \multimap A \otimes !\Gamma$ which begin with a move of A are identical to those of $[\![s', V]\!]$.

Proof A straightforward induction on the derivation of $s, M \Downarrow s', V$. Some cases, like that for application, follow for general reasons such as the comonoidal properties of contraction, whereas others involve examining the behaviour of strategies in detail. For the case of the application rule

$$\frac{s, M \Downarrow s', \lambda x.M' \quad s', M'[N/x] \Downarrow s'', V}{s, MN \Downarrow s'', V}$$

where $\Gamma \vdash M : A \to B$ and $\Gamma \vdash N : A$, we argue as follows. The map

$$!\Gamma \otimes !\Gamma \xrightarrow{[\![MN]\!] \otimes \mathsf{id}} B \otimes !\Gamma$$

is easily seen to be equal to

$$\begin{tikzcd}
!\Gamma \otimes !\Gamma \arrow[d, "{[\![M]\!] \otimes \mathsf{id}}"] \\
(!A \multimap B) \otimes !\Gamma \arrow[d, "\mathsf{id} \otimes \mathsf{con}"] \\
(!A \multimap B) \otimes !\Gamma \otimes !\Gamma \arrow[d, "{\mathsf{id} \otimes [\![N]\!]^{\dagger} \otimes \mathsf{id}}"] \\
(!A \multimap B) \otimes !A \otimes !\Gamma \arrow[d, "\mathsf{ev} \otimes \mathsf{id}"] \\
B \otimes !\Gamma
\end{tikzcd}$$

using the definition of $[\![MN]\!]$ and co-associativity of contraction. Therefore the plays in $[\![s, MN]\!]$ beginning with a move in B involve an interaction in which the initial move in B is copied by ev and two identity strategies to the initial move in $!A \multimap B$, thus beginning a play of $[\![s, M]\!]$. So these plays are determined by the plays of $[\![s, M]\!]$ which begin in $!A \multimap B$. By the inductive hypothesis, these plays are the same as those of $[\![s', \lambda x.M']\!]$. Therefore the plays of $[\![s, MN]\!]$ beginning with a move in B are the same as those of $[\![s', (\lambda x.M')N]\!]$. But $[\![(\lambda x.M')N]\!] = [\![M'[N/x]]\!]$ by the substitution lemma, so the result follows by the inductive hypothesis.

For the case of assignment to a variable,

$$\frac{s, N \Downarrow s', n \quad s', M \Downarrow s'', x}{s, \texttt{assign}\ M\ N \Downarrow (s'' \mid x \mapsto n), \texttt{skip}}$$

we simply unpick the interaction involved in a play of $[\![s, \mathtt{assign}\ M\ \ N]\!]$ beginning with the $\mathtt{run}$ move of $\mathtt{com}$. The $\mathtt{assign}$ strategy responds to this move by interrogating $[\![N]\!]$, which by the inductive hypothesis begins a play identical to that of $[\![s', n]\!]$. Some manipulation as in the case of application shows that the relevant plays of $[\![s, \mathtt{assign}\ M\ \ N]\!]$ are the same as those of $[\![s', \mathtt{assign}\ M\ n]\!]$. After the answer n has been supplied, play continues with $\mathtt{write}(n)$ in the $[\![M]\!]$ part, and again the inductive hypothesis and some manipulation shows that the relevant plays are the same as those of $[\![s'', \mathtt{assign}\ x\ n]\!]$. All that happens in this strategy is that $\mathtt{write}(n)$ is played in the x part fo the store, and then the initial $\mathtt{run}$ is answered with $\mathtt{done}$. Any further play in the $!\Gamma$ component will be copied to and from the newly updated store, so the relevant plays of $[\![s'', \mathtt{assign}\ x\ \ n]\!]$ are the same as those of $[\![(s'' \mid x \mapsto n), \mathtt{skip}]\!]$ as required.

Let us finally consider the case of variable allocation.

$$\frac{(s \mid x \mapsto 0), M \Downarrow (s' \mid x \mapsto n), V}{s, \mathtt{new}\ x\ \mathtt{in}\ M \Downarrow s', V}$$

It is straightforward to show that for any term $\Gamma, x : \mathtt{var} \vdash M$, Γ-store s and natural number k, the map

$$I \xrightarrow{[\![(s \mid x \mapsto k), M]\!]} A \otimes !(\Gamma \times \mathtt{var}) \xrightarrow{\mathsf{id} \otimes !\pi_1} A \otimes !\Gamma$$

is equal to

$$\begin{array}{c} I \\ \big\downarrow{\scriptstyle [\![s]\!]} \\ !\Gamma \\ \big\downarrow{\scriptstyle \mathsf{con}} \\ !\Gamma \otimes !\Gamma \\ \big\downarrow{\scriptstyle \cong} \\ !\Gamma \otimes I \otimes !\Gamma \\ \big\downarrow{\scriptstyle \mathsf{id} \otimes \mathtt{cell}_k \otimes \mathsf{id}} \\ !\Gamma \otimes \mathtt{var} \otimes !\Gamma \\ \big\downarrow{\scriptstyle [\![M]\!] \otimes \mathsf{id}} \\ A \otimes \Gamma. \end{array}$$

Instantiating k with 0, this map is $[\![s, \mathtt{new}\ x\ \mathtt{in}\ M]\!]$, so the relevant plays of this strategy are the same as those of the first map above. By the inductive hypothesis, these are the same as those of $[\![(s \mid x \mapsto n), V]\!]$. Instantiating the diagram above with $k = n$ and $M = V$ gives the result. $\Box$

Corollary 22 If $\vdash M : \mathtt{com}$ is such that $M \Downarrow$ then $[\![M]\!] \neq \bot$.

Again we wish to prove the converse of this result, and again we employ a computability predicate. To define this, we consider what we shall call *split terms*, which are judgements $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ together with a partition of the context $x_1 : A_1, \ldots, x_n : A_n$ into disjoint sets Γ and Δ, where Γ is a var-context and Δ is arbitrary. The idea is that the variables in Γ are those which are bound to storage cells. We will write such a split term as $\Gamma \mid \Delta \vdash M : A$, even though doing so may involve rearranging the order of variables in the original judgement. Split terms with empty Δ part are called *semi-closed*: intuitively, they contain no unbound identifiers. We can now define a predicate of computability on split terms.

Definition

- A semi-closed split term $\Gamma \mid - \vdash M : A$ where A is $\mathtt{com}$ or $\mathtt{exp}$ is computable iff whenever $[\![s]\!] ; [\![M]\!] \neq \bot$, it is the case that $s, M \Downarrow$.
- A semi-closed split term $\Gamma \mid - \vdash M : \mathtt{var}$ is computable iff the split terms $\Gamma \mid - \vdash \mathtt{deref}\ M$ and $\Gamma \mid - \vdash \mathtt{assign}\ M\ n$ are computable for all n.
- A semi-closed split term $\Gamma \mid - \vdash M : A \to B$ is computable iff for all computable $\Gamma \mid - \vdash N : A$ the split term $\Gamma \mid - \vdash MN : B$ is computable.
- An open split term $\Gamma \mid x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ is computable iff for all computable $\Gamma \mid - \vdash N_i : A_i$ the split term $\Gamma \mid - \vdash M[N_1/x_1, \ldots, N_n/x_n]$ is computable.

Lemma 23 All split terms are computable.

Proof The proof is a simple adaptation of that for **PCF**, making use of the soundness result proved above. We first show that all split terms of $\mathbf{IA}_1$, that is, Idealized Algol in which the only allowed uses of $\mathtt{Y}$ are in terms of the form Ω, are computable. This is done by induction on the structure of terms: for every possible splitting of a term, the resulting split term is shown computable. The most interesting case is that of a variable $x : \mathtt{var}$, which may be in the Γ part of the context or the Δ part. In the latter case, we must simply show that $x[N/x]$ is computable for all computable N, which is trivial. In the former case we must verify directly that both $\mathtt{deref}\ x$ and $\mathtt{assign}\ x\ n$ are computable, which follows from the definition of the strategies interpreting these terms.

This result for $\mathbf{IA}_1$ is then extended to the whole of **IA** using syntactic approximants, just as in the proof of Lemma 17. $\Box$

Again the soundness and adequacy results together give us an inequational soundness theorem.

Proposition 24 For all closed **IA** terms M and N, if $[\![M]\!] \subseteq [\![N]\!]$ then $M \lesssim N$.

5.4 Definability

We have seen that extending the games model of **PCF** to one of **IA** requires moving from $\mathcal{C}_{ib}$ to $\mathcal{C}_b$ to accommodate the non-innocent strategy `cell`. We shall now show that in a certain sense, `cell` is a *generic* non-innocent strategy: all strategies in $\mathcal{G}_b$ or $\mathcal{G}$ can be obtained from `cell` and strategies from $\mathcal{G}_{ib}$ or $\mathcal{G}_i$. More precisely, we prove the following factorization theorem.

Proposition 25 Let σ be a strategy for a game A with a countable set of moves, such that whenever $sa \in P_A$ is an odd-length valid position and $\ulcorner sa \urcorner b \in L_A$ for some move b (and justification pointer), the sequence $sab \in P_A$ (where the justification pointer on b is such that $\ulcorner sab \urcorner$ is the same as the sequence $\ulcorner sa \urcorner b$ above). Then there exists an innocent strategy $\tau : !\mathtt{var} \multimap A$ such that

$$\begin{tikzcd} I \arrow[r, "\mathtt{cell}", multimap] \arrow[dr, "\sigma"', multimap] & !\mathtt{var} \arrow[d, "\tau", multimap] \\ & A \end{tikzcd}$$

Moreover, if σ is compact, τ is innocently compact, and if σ is well-bracketed, so is τ. Note that this is a weak orthogonality property in the sense of factorization systems [9].

Proof The idea is that τ simulates σ using its view of the play in A together with (a code for) the full previous history of the play, which it keeps in the variable. Thus we use state to encode history, a standard idea in automata theory; the interesting thing here is that we find a point of contact between machine simulations and factorization systems.

Let the function $\mathsf{code}(-)$ be an injection from valid positions of A to natural numbers such that $\mathsf{code}(\varepsilon) = 0$. Such a function must exist because A has only countably many moves. Note that the P-view of a position s in $!\mathtt{var} \multimap A$ contains the string $\ulcorner s \upharpoonright A \urcorner$. At each O-move a in A, following a previous play s, the strategy τ behaves as follows:

If σ does not have a response at any position of A with player view $\ulcorner sa \upharpoonright A \urcorner$, τ has no response. Otherwise, it reads $\mathsf{code}(t)$ from the variable. Let $b = \sigma(ta)$. If $\ulcorner sa \urcorner b \notin P_A$, τ has no response. Otherwise it writes $\mathsf{code}(tab)$ back into the variable, and then plays b in A.

Note that the assumption on A together with the fact that σ is a valid strategy ensures that τ is well-defined. It is clear that τ is innocent, and that the composite $\mathtt{cell}\,;\tau = \sigma$, as required. Further, if σ is compact, it only has a response b at a finite number of positions ta. In this case, the strategy τ only has a response at a finite number of views, so τ is innocently compact. It is also clear that if σ is well-bracketed then τ must be so as well. $\square$

Lemma 26 Every game of the form $[\![A]\!]$ where A is a type of **IA** satisfies the technical condition in the statement of the above Proposition.

Proof This follows from the fact that for any game $[\![A]\!]$, the valid positions are exactly the legal positions containing at most one initial move. The proof of this fact, which makes use of several technical lemmas from [27], can be found in [5]. $\square$

We can also extend the definability result for **PCF** as follows.

Lemma 27 Every innocently-compact well-bracketed strategy at **IA** type is definable.

The proof of this result is just an extension of the **PCF** case, Proposition 19. In place of the $\mathtt{case}_k$ statements, when a strategy interrogates a `com` type, the $\mathtt{seq}_A$ construct is used—note that its behaviour is exactly that of a "unary case statement". We also make use of `mkvar` to reduce the question of definability for strategies of type $!A \multimap \mathtt{var}$ to those of types $!A \multimap \mathtt{com}$ and $!A \multimap \mathtt{exp}$.

Exercise Write out the details of the above proof. Find an innocently-compact, well-bracketed strategy which is *not* definable without $\mathtt{seq}_{\mathtt{exp}}$, and one not definable without `mkvar`.

We immediately obtain the following definability result.

Proposition 28 Every compact, well-bracketed strategy at **IA** type is definable.

Proof We shall just consider the case of a strategy $\sigma : !A \multimap \mathtt{com}$. (The general case just involves a little more messing around with types, currying, uncurrying and decomposing `var` into `exp` and `com` using `mkvar`.) By Proposition 25, we can write a curried version of σ as

$$I \xrightarrow{\mathtt{cell}} !\mathtt{var} \xrightarrow{\tau} !A \multimap \mathtt{com}$$

where τ is innocently-compact and well-bracketed. By Proposition 28, $\tau = [\![x : \mathtt{var} \vdash \lambda y : A.M : A \to \mathtt{com}]\!]$ for some term M. Hence $\sigma = [\![y : A \vdash \mathtt{new}\ x\ \mathtt{in}\ M]\!]$ as required. $\square$

Full abstraction for the model in $\mathcal{C}_b$ now follows exactly as in the case of **PCF**.

Theorem 29 For all closed terms M and N of **IA**,

$$[\![M]\!] \lesssim_b [\![N]\!] \iff M \sqsubseteq_{\sim} N.$$

6 PCF with control

Just as important as the issue of state in programming languages is that of control. Almost all languages in use offer facilities for manipulating the flow of control, be they goto statements and labels, exceptions and other error trapping mechanisms, or the various call/cc and catch/throw operators which appear in languages with a functional core. All these constructs give the programmer the ability to program non-local jumps which are not present in simple functional languages in the style of **PCF**, or in Idealized Algol.

We will show in this section that the addition of a simple control operator to **PCF** corresponds precisely to the violation of the bracketing condition on strategies, just as the addition of state corresponds to the violation of innocence. The control operator we add is a very simple catch construct, similar to that of [10], so our extended language is (a minor variant of) the language called **SPCF** in that paper; we will use the same name.

We add to **PCF** a family of control operators called $\mathtt{catch}_k$. The typing rule is as follows.

$$\frac{\Gamma, x_1 : \mathtt{exp}, \ldots, x_k : \mathtt{exp} \vdash M : \mathtt{exp}}{\Gamma \vdash \mathtt{catch}_k\ x_1, \ldots, x_k\ \mathtt{in}\ M}$$

This operation binds $x_1, \ldots, x_k$ in M.

Using catch allows early exit from expressions. Intuitively, when the term M tries to evaluate variable x_i, the term $\mathtt{catch}\ x_1, \ldots, x_k\ \mathtt{in}\ M$ immediately terminates, returning $i - 1$. Should M terminate with some n without using any of the x_i, $\mathtt{catch}\ x_1, \ldots, x_k\ \mathtt{in}\ M$ returns $n + k$.

6.1 Operational semantics

Unfortunately, the big-step style of operational semantics we have used so far is not informative enough to allow us to extend it to **SPCF**. We therefore use a small-step style of semantics. We shall also make use of the notion of evaluation context. Intuitively, an evaluation context is a context in which the single hole is "in the place which will be evaluated next". Formally, evaluation contexts are defined by the following grammar.

$$E ::= - \mid EM \mid \mathtt{succ}\ E \mid \mathtt{pred}\ E \mid \mathtt{cond}\ E\ N_1\ N_2 \mid \mathtt{catch}_k\ x_1, \ldots, x_k\ \mathtt{in}\ E.$$

Hand in hand with evaluation contexts goes the notion of of redex. A redex is a term in which a computation step is about to happen; redexes are defined below.

$$\begin{array}{rcl} R & ::= & (\lambda x.M)N \mid \mathtt{succ}\ n \mid \mathtt{pred}\ n \mid \mathtt{cond}\ n\ N_1\ N_2 \mid \mathtt{Y}\ M \\ & \mid & \mathtt{catch}\ x_1, \ldots, x_k\ \mathtt{in}\ n \mid \mathtt{catch}\ x_1, \ldots, x_k\ \mathtt{in}\ E[x_i] \end{array}$$

where in the last case, the occurrence of x_i in $E[x_i]$ is assumed free.

Lemma 30 Any term of **SPCF** is either a canonical form or can be written uniquely as $E[M]$, where E is an evaluation context and M is either a free variable or a redex.

Proof An easy induction over the structure of terms. $\square$

We now give the operational semantics in two stages. First we say how redexes reduce.

$$
\begin{array}{rcl}
(\lambda x.M)N & \to & M[N/x] \\
\mathtt{succ}\, n & \to & n+1 \\
\mathtt{pred}\, 0 & \to & 0 \\
\mathtt{pred}\, n+1 & \to & n \\
\mathtt{cond}\ 0\ N_1\ N_2 & \to & N_1 \\
\mathtt{cond}\ n+1\ N_1\ N_2 & \to & N_2 \\
\mathtt{Y}\ M & \to & M(\mathtt{Y}\ M) \\
\mathtt{catch}\, x_1, \ldots, x_k\ \mathtt{in}\ n & \to & n+k \\
\mathtt{catch}\, x_1, \ldots, x_k\ \mathtt{in}\ E[x_i] & \to & i-1
\end{array}
$$

Again, in the last rule we assume the occurrence of x_i in the hole of $E[x_i]$ is free. We now define the reduction $M \mapsto N$ of arbitrary terms by:

$$
\frac{M \to N}{E[M] \mapsto E[N].}
$$

Note that this operational semantics reduces open terms as well as closed terms. For a closed term M, we write $M \Downarrow V$ if $M \mapsto^* V$ where V is in canonical form. (Here $\mapsto^*$ denotes the reflexive transitive closure of $\mapsto$.)

Exercise Show that for **PCF** terms, the relation $M \Downarrow V$ defined here coincides with that of Section 4.1.

6.2 Denotational semantics

We now extend the game semantics of **PCF** to **SPCF** by giving strategies interpreting the $\mathtt{catch}_k$ constructs. These strategies are innocent but not well-bracketed, so we obtain models of **SPCF** in $\mathcal{C}_i$ and $\mathcal{C}$.

We shall just describe the strategy $c_2 : (\mathbf{N} \times \mathbf{N} \to \mathbf{N}) \to \mathbf{N}$ used to interpret

$\mathtt{catch}_2$, by giving its typical plays, which are shown below.

$$
\begin{array}{ccccccc}
!(!\mathbf{N} & \otimes & !\mathbf{N} & \multimap & \mathbf{N}) & \longrightarrow\!\circ & \mathbf{N} \\
 & & & & & & q \\
 & & & & q & & \\
q & & & & & & \\
 & & & & & & 0 \\
\hline
 & & & & & & q \\
 & & & & q & & \\
 & & q & & & & \\
 & & & & & & 1 \\
\hline
 & & & & & & q \\
 & & & & q & & \\
 & & & & n & & \\
 & & & & & & n+2 \\
\hline
\end{array}
$$

In fact these three plays are all the maximal views of the strategy c_2. Given an **SPCF** term $\Gamma, x_1 : \mathtt{exp}, x_2 : \mathtt{exp} \vdash M : \mathtt{exp}$ whose interpretation is a map $[\![M]\!] : \Gamma \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ in $\mathcal{C}_i$ or $\mathcal{C}$, by currying and composing with c_2 we obtain $[\![\mathtt{catch}\, x_1, x_2 \;\mathtt{in}\; M]\!]$:

$$\Gamma \xrightarrow{\Lambda([\![M]\!])} \mathbf{N} \times \mathbf{N} \Rightarrow \mathbf{N} \xrightarrow{c_2} \mathbf{N}.$$

6.3 Soundness and Adequacy

We now set about the business of showing that our model of **SPCF** is sound and adequate, as we have done for **PCF** and **IA** before. First a technical lemma which essentially says that evaluation contexts are strict with respect to any term which may fill the hole.

Lemma 31 For any term $\Gamma, x : A \vdash E[x] : B$, the strategy

$$[\![E[x]]\!] : !\Gamma \otimes !A \multimap B$$

responds to the initial question q of B with the initial question q' of $!A$.

Proof A simple induction over the structure of evaluation contexts. $\Box$

We can now prove a soundness result.

Lemma 32 For any **SPCF** term M, if $M \mapsto N$ then $[\![M]\!] = [\![N]\!]$.

Proof We shall just show that for any redex R, if $R \to R'$ then $[\![R]\!] = [\![R']\!]$. The rest follows from the compositionality of our semantics. For all redexes apart from those of `catch`, the required result holds for the same reasons that the **PCF** model is sound, so we shall not go into further detail. The first `catch` reduction

$$\mathtt{catch}\, x_1, \ldots, x_k \;\mathtt{in}\; n \to n + k$$

can be checked straightforwardly. For the second reduction,

$$\texttt{catch}\, x_1, \ldots, x_k\ \texttt{in}\ E[x_i] \rightarrow i - 1$$

we use Lemma 31, which tells us that the map

$$\Lambda([\![E[x_i]]\!]) : \Gamma \rightarrow \mathbf{N}_1 \times \cdots \times \mathbf{N}_k \Rightarrow \mathbf{N}$$

responds to the initial question in $\mathbf{N}$ with that of $\mathbf{N}_i$. It is then straightforward to check that $[\![\texttt{catch}\, x_1, \ldots, x_k\ \texttt{in}\ E[x_i]]\!] = [\![i - 1]\!]$. □

This Lemma implies that if $M \Downarrow n$ then $[\![M]\!] = [\![n]\!] \neq \bot$. As usual we require *adequacy*, that is, the converse of this result, and as usual a computability argument is employed. The proof we shall give differs from those for **PCF** and **IA**, and is adapted from that found in [10].

Let **SPCF**$_1$ be the restricted subset of **SPCF** in which the only allowed terms of the form $\texttt{Y}\ M$ are the Ω terms. As usual we shall prove our result first for **SPCF**$_1$ before extending it to full **SPCF**. Our predicate of computability is defined as follows.

Definition

- A (possibly open) term $M : \texttt{exp}$ is computable iff either $M \mapsto^* n$, $M \mapsto^* E[x]$ for some free variable x, or $M \mapsto^* E[\Omega]$.
- A (possibly open) term $M : A \rightarrow B$ is computable iff $MN : B$ is computable for all computable $N : A$.

Notice that this definition does not involve the semantics at all; however, it is not so different from the other definitions, because Lemma 31 implies that $[\![E[\Omega]]\!] = \bot$.

Lemma 33 If $M \mapsto M'$ and M' is computable, then M is computable.

Proof By induction on type. The case of $\texttt{exp}$ is trivial. In the case $M \mapsto M' : A \rightarrow B$, we must show that $MN : B$ is computable for all computable $N : A$. Since $M \mapsto M'$, we know that $M = E[R]$ for some evaluation context E and redex R, and that $R \rightarrow R'$ and $M' = E[R']$. But then $MN = E[R]N$, and $E[-]N$ is an evaluation context, so $MN \mapsto E[R']N = M'N$. We know $M'N$ is computable, so by the inductive hypothesis, MN is computable. □

We can now prove the main lemma for adequacy.

Lemma 34 For any term $x_1 : A_1, \ldots, x_k : A_k \vdash M$ of **SPCF**$_1$ and any computable (and possibly open) $N_1 : A_1, \ldots, N_k : A_k$, $M[N_1/x_1, \ldots, N_k/x_k]$ is computable.

Proof By induction on the structure of M. The cases of variables, numerals and the term Ω are all trivial. Those of $\texttt{succ}$, $\texttt{pred}$ and $\texttt{cond}$ are all similar, so we give the argument for $\texttt{succ}$ as an illustration.

If $M = \texttt{succ}\ M'$ then $M[\vec{N}/\vec{x}] = \texttt{succ}\ M'[\vec{N}/\vec{x}]$, and $M'[\vec{N}/\vec{x}]$ is computable, by the inductive hypothesis. One of the following therefore holds.

- $M'[\vec{N}/\vec{x}] \mapsto^* n$, in which case $\texttt{succ}\ M'[\vec{N}/\vec{x}] \mapsto^* n+1$.
- $M'[\vec{N}/\vec{x}] \mapsto^* E[x]$, in which case $\texttt{succ}\ M'[\vec{N}/\vec{x}] \mapsto^* \texttt{succ}\ E[x]$.
- $M'[\vec{N}/\vec{x}] \mapsto^* E[\Omega]$, in which case $\texttt{succ}\ M'[\vec{N}/\vec{x}] \mapsto^* \texttt{succ}\ E[\Omega]$.

In the first case, we are done. In the second and third cases, we just need to observe that $\texttt{succ}\ E[-]$ is an evaluation context, so we are done.

The case of application follows directly from the inductive hypothesis. For abstraction, when $M = \lambda x.M'$, we have $M[\vec{N}/\vec{x}] = \lambda x.M'[\vec{N}/\vec{x}]$. Letting N be any computable term of the right type, we must show that $(\lambda x.M'[\vec{N}/\vec{x}])N$ is computable. But $(\lambda x.M'[\vec{N}/\vec{x}])N \mapsto M'[\vec{N}/\vec{x}][N/x]$ which is computable by the inductive hypothesis, and an appeal to Lemma 33 gives the required result.

Finally, we consider the case $M = \texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ M'$. Now $M[\vec{N}/\vec{x}] = \texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ M'[\vec{N}/\vec{x}]$ and we know $M'[\vec{N}/\vec{x}]$ is computable by the inductive hypothesis. Again there are three cases: $M'[\vec{N}/\vec{x}]$ reduces either to n, $E[z]$ for some free variable z, or $E[\Omega]$. In the first and third of these cases, $\texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ M'[\vec{N}/\vec{x}]$ is immediately seen to be computable. In the second case, if z is not one of the x_i, we have $\texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ M'[\vec{N}/\vec{x}] \mapsto^* \texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ E[z]$ and z is free, so since $E' = \texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ E[-]$ is an evaluation context, we are done. Finally, if z is some x_i, then

$$\texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ M'[\vec{N}/\vec{x}] \mapsto^* i-1,$$

completing the proof. □

Corollary 35 All terms of $\mathbf{SPCF}_1$ are computable.

Proof It is clear that variables are computable, so given any term $x_1,\ldots,x_n \vdash M : A$ of $\mathbf{SPCF}_1$, the term $M[x_1/x_1,\ldots,x_n/x_n]$ is computable by the above Lemma. But this term is the same as M. □

We lift this result to full **SPCF** using the notion of syntactic approximant as before. This time, the important lemma is the following.

Lemma 36 If $M \prec N$ and M is not of the form $E[\Omega]$, then if $M \mapsto M'$, there exists an N' such that $M' \prec N'$ and $N \mapsto^* N'$.

Proof Induction on the derivation of $M \prec N$. In the case of a reduction

$$M = \texttt{catch}\ x_1,\ldots,x_k\ \texttt{in}\ E[x_i] \to i-1$$

we need to show that if $E[x] \prec N$ then $N \mapsto^* E'[x]$ for some E'; but this is straightforward. □

Proposition 37 If $\vdash M : \texttt{exp}$ is a program and $[\![M]\!] \neq \bot$ then $M\Downarrow$.

Proof If $[\![M]\!] \neq \bot$, it cannot be the case that $M \mapsto^* E[\Omega]$ since $[\![\Omega]\!] = \bot$ and evaluation contexts are strict. If M is a term of **SPCF**$_1$, then by Lemma 34, $M \mapsto^* n$ for some n, so $M\Downarrow$. For an arbitrary term M, as usual we have

$$[\![M]\!] = \bigcup_n [\![M_n]\!]$$

where M_n is M with all subterms $\texttt{Y}\ N$ replaced by $\texttt{Y}^n\ N_n$; recall that $M_n \prec M$. For some n, we have $[\![M_n]\!] \neq \bot$, so $M_n\Downarrow$ by the argument above. Then by Lemma 36, we also have $M\Downarrow$. $\square$

Proposition 38 If $[\![M]\!] \subseteq [\![N]\!]$ for closed **SPCF** terms M and N, then $M \precsim N$.

6.4 Definability

We can now prove definability for the model of **SPCF** in the category $\mathcal{C}_i$. As was the case for **IA**, the proof makes use of a factorization theorem which removes violations of the bracketing condition from an arbitrary compact innocent strategy. The technique behind this factorization is due to Laird [24].

Lemma 39 Let A be an **SPCF** type and $\sigma : A$ an innocently-compact strategy. There exists a natural number k and an innocently-compact, well-bracketed strategy

$$\tau : !((\mathbf{N}_1 \times \cdots \times \mathbf{N}_k \Rightarrow \mathbf{N}_{k+1}) \Rightarrow \mathbf{N}_{k+2}) \multimap \mathbf{N}$$

such that $c_k^\dagger ; \tau = \sigma$, where c_k is the strategy used in the interpretation of $\texttt{catch}_k$, viewed as a map

$$c_k : I \multimap (\mathbf{N}_1 \times \cdots \times \mathbf{N}_k \Rightarrow \mathbf{N}_{k+1}) \Rightarrow \mathbf{N}_{k+2}.$$

Proof The strategy τ "factors out" all violations of the bracketing condition from σ. Suppose one of the plays of σ is of the form

$$\ldots q \ldots i - 1$$

where the answer $i - 1$ is a P-move violating the bracketing condition. Choose k large enough that any answer σ may give to this q is smaller than k. The idea behind the factorization to remove this violation is suggested by the following diagram.

$$
\begin{array}{cccccccl}
((\mathbf{N}_1 \times \cdots \times \mathbf{N}_k) & \Rightarrow & \mathbf{N}_{k+1}) & \Rightarrow & \mathbf{N}_{k+2} & \rightarrow & A & \\
 & & & & & & \vdots & \Big\} \text{ play as } \sigma \\
 & & & & & & q & \\
 & & & & q_{k+2} & & & \\
 & & q_{k+1} & & & & & \\
 & & & & & & \vdots & \Big\} \text{ play as } \sigma \\
q_i & & & & & & & \\
 & & & & i-1 & & & \\
 & & & & & & i-1 & \\
 & & & & & & \vdots & \Big\} \text{ play as } \sigma
\end{array}
$$

The strategy τ plays exactly as σ would play in A, until O plays the question q, whereupon τ asks the question q_{k+2} on the left hand side. If O is playing according to c_k, the next move will be q_{k+1} as shown. Now τ switches back to A and continues playing as σ would play, until σ would give the answer $i-1$ to q. At this point τ asks the question q_i in $\mathbf{N}_i$, justified by the q_{k+1} previously played by O. If O is playing according to c_k, the response is to play $i-1$, answering q_{k+2}. Now τ can copy this answer to A, answering q without violating the bracketing condition, because of the intervening pair of moves

$$q_{k+2} \ldots i - 1.$$

Since τ is intended to be an innocent strategy, we must add to it some behaviours for the case when O does not play according to c_k. Whenever O answers q_{k+2} with some n, τ copies this answer to A, answering q. If O answers the question q_i, τ has no response. Playing according to this strategy, after an O-move in A, the P-view is always a P-view of a position in A reachable by σ, with the two moves $q_{k+2} \cdot q_{k+1}$ inserted after q if it appears in this view. The strategy τ responds at any such view with the move that σ would have played. It is straightforward to check that τ is a well-defined, innocently-compact strategy and that there are strictly fewer views in τ which violate the bracketing condition than there are in σ. It is also clear that $c_k^\dagger ; \tau = \sigma$. To complete the proof we simply apply the above factorization repeatedly to remove all violations of the bracketing condition. $\square$

Just as in the case of **IA**, this factorization leads to definability and full abstraction.

Proposition 40 Let $\sigma : A_1 \times \cdots \times A_n \rightarrow A$ be an innocently-compact map in $\mathcal{C}_i$, where the A_i and A are types of **SPCF**. There exists an **SPCF** term $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ such that $\sigma = [\![M]\!]$.

Proof By Lemma 39, there exists an innocently-compact well-bracketed strategy

$$\tau : ((\mathbf{N}_1 \times \cdots \times \mathbf{N}_k \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}) \rightarrow (A_1 \times \cdots \times A_n \Rightarrow A)$$

such that $c_k \fatsemi \tau$ is a suitably curried version of σ. If we then curry the type on the domain of τ so that it is a **PCF** type, then by Proposition 19,

$$\tau = [\![x : (\mathtt{exp}_1 \to \cdots \to \mathtt{exp}_k \to \mathtt{exp}) \to \mathtt{exp} \vdash \lambda a_1, \ldots, \lambda a_n : M]\!].$$

Then

$$\sigma = [\![a_1 : A_1, \ldots, a_n : A_n \vdash (\lambda x.M)(\lambda f.\mathtt{catch}\ x_1, \ldots, x_k\ \mathtt{in}\ f x_1 \ldots x_k)]\!]$$

where f has type $\mathtt{exp}_1 \to \cdots \to \mathtt{exp}_k \to \mathtt{exp}$. $\square$

As before, this definability result leads to full abstraction in the category which results from quotienting with respect to the intrinsic preorder.

Theorem 41 For closed **SPCF** terms M and N of the same type,

$$[\![M]\!] \lesssim_i [\![N]\!] \iff M \sqsubseteq\!\!\!\!\sim N.$$

References

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

[2] S. Abramsky. Axioms for full abstraction and full completeness. In G. Plotkin, C. Sterling, and M. Tofte, editors, *Essays in Honour of Robin Milner*. MIT Press, to appear.

[3] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1998.

[4] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543 – 574, June 1994. Also appeared as Technical Report 92/24 of the Department of Computing, Imperial College of Science, Technology and Medicine.

[5] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In O'Hearn and Tennent [32], pages 297–329 of volume 2.

[6] S. Abramsky and G. McCusker. Call-by-value games. In M. Nielsen and W. Thomas, editors, *Computer Science Logic: 11th International Workshop Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

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

[8] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.

[9] F. Borceux. *Handbook of Categorical Algebra, volume 1*. Cambridge University Press, 1994.

[10] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.

[11] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

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

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[14] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. Accepted for publication in Information and Computation, 1997.

[15] R. Kelsey, W. Clinger, and J. Rees. The revised^5 report on the algorithmic language Scheme. 1998.

[16] S. C. Kleene. Recursive functionals and quantifiers of finite types I. *Transactions of the AMS*, 91:1–52, 1959.

[17] S. C. Kleene. Turing-machine computable functionals of finite types I. In P. Suppes, editor, *Proc. 1960 Congress of Logic, Methodology and the Philosophy of Science*, pages 38–45. North-Holland, Amsterdam, 1962.

[18] S. C. Kleene. Turing-machine computable functionals of finite types II. *Proceedings of the London Mathematical Society*, 12 (3rd series):245–258, 1962.

[19] S. C. Kleene. Recursive functionals and quantifiers of finite types II. *Transactions of the AMS*, 108:106–142, 1963.

[20] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited I. In J. E. Fenstad, R. O. Gandy, and G. E. Sacks, editors, *Generalized Recursion Theory II*, pages 185–222. North-Holland, Amsterdam, 1978.

[21] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited II. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 1–29. North-Holland, Amsterdam, 1980.

[22] S. C. Kleene. Recursive functionals and quantifiers of finite types revisited III. In G. Metakides, editor, *Patras Logic Symposium*, pages 1–40. North-Holland, Amsterdam, 1982.

[23] S. C. Kleene. Unimonotone functions of finite types (recursive functionals and quantifiers of finite types revisited IV). In A. Nerode and R. A. Shore, editors, *Recursion Theory*, pages 119–138. American Mathematical Society, Providence, Rhode Island, 1985.

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

[25] R. Loader. Finitary PCF is not decidable. Unpublished manuscript, 1996.

[26] G. McCusker. Games and full abstraction for FPC (full version). Submitted for publication, 1996.

[27] G. McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. Distinguished Dissertations in Computer Science. Springer-Verlag, 1998.

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

[29] R. Milner. Functions as processes. In *Proceedings of ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.

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

[31] H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, Lecture notes in Computer Science. Springer, 1994.

[32] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages*. Birkhaüser, 1997.

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

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

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