Playful, streamlike computation

Pierre-Louis Curien (CNRS & Université Paris VII)

April 10, 2003

Abstract

We offer a short tour into the interactive interpretation of sequential programs. We emphasize *streamlike* computation – that is, computation of successive bits of information upon request. The core of the approach surveyed here dates back to the work of Berry and the author on *sequential algorithms on concrete data structures* in the late seventies, culminating in the design of the programming language CDS, in which the semantics of programs of any type can be explored interactively. Around one decade later, two major insights of Cartwright and Felleisen on one hand, and of Lamarche on the other hand gave new, decisive impulses to the study of sequentiality. Cartwright and Felleisen observed that sequential algorithms give a direct semantics to control operators like `call-cc` and proposed to include explicit errors both in the syntax and in the semantics of the language PCF. Lamarche (unpublished) connected sequential algorithms to linear logic and games. The successful program of games semantics has spanned over the nineties until now, starting with syntax-independent characterizations of the term model of PCF by Abramsky, Jagadeesan, and Malacaria on one hand, and by Hyland and Ong on the other hand.

Only a basic acquaintance with λ-calculus, domains and linear logic is assumed in sections 1 through 3.

1 Prologue: playing with Böhm trees

We first make some preparations. For self-containedness, we briefly recall the relevant notions. The syntax of the untyped λ-calculus (λ-calculus for short) is given by the following three constructions: a *variable* x is a λ-term, if M and N are λ-terms, then the *application* MN is a λ-term, and if M is a term then the *abstraction* $\lambda x.M$ is a term. Usual abbreviations are $\lambda x_1 x_2.M$ for $\lambda x_1.(\lambda x_2.M)$, and MN_1N_2 for $(MN_1)N_2$, and similarly for n-ary abstraction and application. A more macroscopic view is quite useful: it is easy to check that any λ-term has exactly one of the following two forms:

$$\begin{array}{ll} (n \geq 1, p \geq 1) & \lambda x_1 \cdots x_n . x M_1 \cdots M_p \\ (n \geq 0, p \geq 1) & \lambda x_1 \cdots x_n . (\lambda x . M) M_1 \cdots M_p \ . \end{array}$$

The first form is called a *head normal form* (hnf), while the second exhibits the *head redex* $(\lambda x.M)M_1$. The following easy property justifies the name of head normal form: any reduction sequence starting from a hnf $\lambda x_1 \cdots x_n.xM_1 \cdots M_p$ consists of an interleaving of independent reductions of $M_1, \ldots, M_p$. More precisely, we have:

$$\begin{array}{l} (\lambda x_1 \cdots x_n . x M_1 \cdots M_p \to^* P) \Rightarrow \\ \exists N_1, \ldots N_p \ \left\{ \begin{array}{l} P = \lambda x_1 \cdots x_n . x N_1 \cdots N_p \text{ and} \\ \forall\, i \leq p \ \ M_i \to^* N_i \ . \end{array} \right. \end{array}$$

Here, reduction means the replacement in any term of a sub-expression of the form $(\lambda x.M)N$, called a β-redex, by $M[x \leftarrow N]$. A normal form is a term that contains no β-redex, or equivalently that contains no head redex. Hence the syntax of normal forms is given by the following two constructions: a variable x is a normal form, and if $M_1, \ldots, M_p$ are normal forms, then $\lambda x_1 \cdots x_n.xM_1 \cdots M_p$ is a normal form.

Now, we are ready to play. Consider the following two normal forms:

$$M = zM_1M_2(\lambda z_1 z_2 . z_1 M_3 M_4) \qquad N = \lambda x_1 x_2 x_3 . x_3 (\lambda y_1 y_2 . y_1 N_1) N_2 \ .$$

The term $M[z \leftarrow N] = NM_1M_2(\lambda z_1z_2.z_1M_3M_4)$ is not a normal form anymore, and can be β-reduced as follows:

$$\begin{array}{rcl} NM_1M_2(\lambda z_1 z_2 . z_1 M_3 M_4) & \to & (\lambda z_1 z_2 . z_1 M_3 M_4)(\lambda y_1 y_2 . y_1 N_1') N_2' \\ & \to & (\lambda y_1 y_2 . y_1 N_1') M_3' M_4' \\ & \to & M_3' N_1'' \ , \end{array}$$

where N_1', etc... are suitable substitution instances of N_1 etc... But there is a more geometric way of describing the *interaction* of M and N. First, we represent M and N explicitly as trees (read from left to right), as follows:

$$z \left\{ \begin{array}{l} M_1 \\ M_2 \\ \lambda z_1 z_2 .\ z_1 \left\{ \begin{array}{l} M_3 \\ M_4 \end{array} \right. \end{array} \right. \qquad \lambda x_1 x_2 x_3 .\ x_3 \left\{ \begin{array}{l} \lambda y_1 y_2 .\ y_1 \left\{ \ N_1 \right. \\ N_2 \end{array} \right.$$

Then we represent computation as the progression of two tokens in the two trees. Initially, the tokens are at the root (we use underlining to indicate the location of the tokens):

$$\underline{z}\left\{\begin{array}{l} M_1 \\ M_2 \\ \lambda z_1 z_2.\, z_1 \left\{\begin{array}{l} M_3 \\ M_4 \end{array}\right. \end{array}\right. \qquad \underline{\lambda x_1 x_2 x_3.}\, x_3 \left\{\begin{array}{l} \lambda y_1 y_2.\, y_1 \left\{\ N_1\right. \\ N_2 \end{array}\right.$$

We then notice that the token in M has a choice of where to move right, while the one on the right has no choice. So let us take the latter as pilot:

$$\underline{z}\left\{\begin{array}{l} M_1 \\ M_2 \\ \lambda z_1 z_2.\, z_1 \left\{\begin{array}{l} M_3 \\ M_4 \end{array}\right. \end{array}\right. \qquad \lambda x_1 x_2 x_3.\, \underline{x_3} \left\{\begin{array}{l} \lambda y_1 y_2.\, y_1 \left\{\ N_1\right. \\ N_2 \end{array}\right.$$

Here, $\underline{x_3}$ reads as "take the third branch" (in M):

$$z\left\{\begin{array}{l} M_1 \\ M_2 \\ \underline{\lambda z_1 z_2.}\, z_1 \left\{\begin{array}{l} M_3 \\ M_4 \end{array}\right. \end{array}\right. \qquad \lambda x_1 x_2 x_3.\, \underline{x_3} \left\{\begin{array}{l} \lambda y_1 y_2.\, y_1 \left\{\ N_1\right. \\ N_2 \end{array}\right.$$

Notice that we are now in a situation where the token in M knows how to move next, while the one in N has a choice. We take M as pilot:

$$z\left\{\begin{array}{l} M_1 \\ M_2 \\ \lambda z_1 z_2.\, \underline{z_1} \left\{\begin{array}{l} M_3 \\ M_4 \end{array}\right. \end{array}\right. \qquad \lambda x_1 x_2 x_3.\, \underline{x_3} \left\{\begin{array}{l} \lambda y_1 y_2.\, y_1 \left\{\ N_1\right. \\ N_2 \end{array}\right.$$

We read z_1 as "take the first branch" (in N):

$$z\left\{\begin{array}{l} M_1 \\ M_2 \\ \lambda z_1 z_2.\, \underline{z_1} \left\{\begin{array}{l} M_3 \\ M_4 \end{array}\right. \end{array}\right. \qquad \lambda x_1 x_2 x_3.\, x_3 \left\{\begin{array}{l} \underline{\lambda y_1 y_2.}\, y_1 \left\{\ N_1\right. \\ N_2 \end{array}\right.$$

The following steps are:

$$z \begin{cases} M_1 \\ M_2 \\ \lambda z_1 z_2.\ \underline{z_1} \begin{cases} M_3 \\ M_4 \end{cases} \end{cases} \qquad \lambda x_1 x_2 x_3.\ x_3 \begin{cases} \lambda y_1 y_2.\ \underline{y_1} \left\{\ N_1 \right. \\ N_2 \end{cases}$$

$$z \begin{cases} M_1 \\ M_2 \\ \lambda z_1 z_2.\ z_1 \begin{cases} \underline{M_3} \\ M_4 \end{cases} \end{cases} \qquad \lambda x_1 x_2 x_3.\ x_3 \begin{cases} \lambda y_1 y_2.\ \underline{y_1} \left\{\ N_1 \right. \\ N_2 \end{cases}$$

We leave it to the reader to check that these steps follow closely the sequence of β-reductions given above. The graphical presentation enhances alternation. The tokens' moves alternate between M and N. There are two sorts of moves: variables (like z), and (bunches of) abstractions (like $\lambda z_1 z_2.$). We call these moves Player's moves and Opponent's moves, respectively. We can view an Opponent's move as the question "what is the head variable of the term rooted here?", and a Player's move as the answer to this question. So we see the computation as a progression of alternating moves describing a path in M (and in N):

$$\underline{z} \begin{cases} M_1 \\ M_2 \\ \underline{\lambda z_1 z_2.}\ \underline{z_1} \begin{cases} \underline{M_3} \\ M_4 \end{cases} \end{cases} \qquad \underline{\lambda x_1 x_2 x_3.}\ \underline{x_3} \begin{cases} \underline{\lambda y_1 y_2.}\ \underline{y_1} \left\{\ N_1 \right. \\ N_2 \end{cases}$$

Our example is actually too simple. The general mechanism needs an explicit manipulation of pointers, when (unlike in the example) a variable is not bound by the immediate bunch of λ's above. We refer the interested reader to [15, **?**], where this machinery is described for a larger class of trees with pointers – called abstract Böhm trees –, of which Böhm trees are an example. Our main point here was to highlight interaction: M and N are pilot in turn and tell the other which branch to choose.

Suppose now that $M_3 = \lambda u.tM_5M_6$, where t is a free variable of $M[z \leftarrow N]$. Then, looking back at the sequence of β-reductions, we reach a head normal form:

$$M_3' N_1'' = (\lambda u.tM_5' M_6')N_1'' \quad \rightarrow \quad tM_5'' M_6'' \ .$$

And, in geometrical form:

$$z \left\{ \begin{array}{l} M_1 \\ M_2 \\ \lambda z_1 z_2 .\, z_1 \left\{ \begin{array}{l} \lambda u .\, \underline{t} \left\{ \begin{array}{l} M_5 \\ M_6 \end{array} \right. \\ M_4 \end{array} \right. \end{array} \right. \qquad \lambda x_1 x_2 x_3 .\, x_3 \left\{ \begin{array}{l} \lambda y_1 y_2 .\, \underline{y_1} \left\{ \; N_1 \right. \\ N_2 \end{array} \right.$$

Note here that N cannot help to choose the next move in M. The machinery stops here. After all, most functional programming languages stop evaluation on (weak) head normal forms. But what about getting the full normal form, i.e., computing M_5'' and M_6''? The interactive answer to this question is: by exploration of branches, on demand, or in a streamlike manner. The machine displays t as the head variable of the normal form of $M[z \leftarrow N]$. Now, you, the Opponent, can choose which of the branches below t to explore, and then the machine will restart until it reaches a head normal form. For example, if you choose the first branch, then you will eventually reach the head variable of M_5''. This is called streamlike, because that sort of mechanism has been first analysed for infinite lists built progressively. A λ-term too has a "potentially infinite normal form": it's Böhm tree.

This prologue served the purpose of introducing some keywords, such as interactivity, playful interpretation, streamlike computation. We now start from the beginning.

2 Introduction

Scott's and Plotkin's denotational semantics takes its roots in recursion theory. It is worth recalling here the statement of Rice's theorem. This theorem asserts a property of recursively enumerable (r.e.) sets of partial recursive (p.r.) functions, defined through a fixed enumeration (ϕ_n) of the p.r. functions (i.e. ϕ is a surjection from ω – the set of natural numbers – to $\omega \rightharpoonup \omega$, using $\rightharpoonup$ for sets of partial functions). Let $PR \subseteq \omega \rightharpoonup \omega$ denote the set of p.r. functions. A subset $A \subseteq PR$ is called r.e. if $\{n \mid \phi_n \in A\}$ is r.e. in the usual sense. The theorem asserts that if A is r.e. and if $f \in A$, then there exists a finite approximation g of f such that $g \in A$. That g is an approximation of f means that f is an extension of g, i.e., the domain on which the partial function f is defined, or domain of definition of f, contains that of g and f and g coincide where they are both defined. A simpler way of saying this is that the graph of g is contained in the graph of f. Moreover, the domain of definition of g is finite. Rice's theorem is about an intrinsic continuity property in the realm of p.r. functions. It highlights the (complete) partial order structure of $\omega \rightharpoonup \omega$, and in particular the presence of a bottom element $\perp$ in this partial order: the everywhere undefined function.

Certainly, one of the key departure points taken by Scott was to take $\perp$ seriously. Once this element is part of the picture, one takes a new look at some basic functions. Take the booleans, for example. In Scott's semantics, this is not the set $\{tt, ff\}$, but the set $\{\perp, tt, ff\}$ ordered as follows: $x \leq y$ if and only if $x = y$ or $x = \perp$ (this is called *flat* ordering). Take now the good old disjunction function $or : Bool \times Bool \to Bool$. It gives rise to four different functions over the flat domain version of $Bool$ (the specifications below can be completed to full definitions by monotonicity):

$$\begin{array}{l}
por(\perp, tt) = tt \\
por(tt, \perp) = tt \\
por(\perp, ff) = \perp \\
por(ff, \perp) = \perp \\
por(ff, ff) = ff
\end{array}$$

$$\begin{array}{lll}
lor(\perp, y) = \perp & \qquad & ror(\perp, tt) = tt \\
lor(tt, \perp) = tt & & ror(x, \perp) = \perp \\
lor(ff, y) = y & & ror(x, ff) = x
\end{array}$$

$$\begin{array}{l}
sor(\perp, tt) = \perp \\
sor(tt, \perp) = \perp \\
sor(\perp, ff) = \perp \\
sor(ff, \perp) = \perp \\
sor(ff, tt) = tt \\
sor(tt, ff) = tt \\
sor(tt, tt) = tt \\
sor(ff, ff) = ff
\end{array}$$

It should be clear that lor and ror are computed by programs of the following shape, respectively:

$$\lambda xy.\, \mathtt{if}\ x = tt \begin{cases} \mathtt{then}\ tt \\ \mathtt{else}\ \mathtt{if}\ y = \cdots \end{cases} \qquad \lambda xy.\, \mathtt{if}\ y = tt \begin{cases} \mathtt{then}\ tt \\ \mathtt{else}\ \mathtt{if}\ x = \cdots \end{cases}$$

On the other hand, it should be intuitively clear that no sequential program of the same sort can compute por, because a sequential program will

- either start by examining one of the arguments, say x, in which case it can't output anything before a value for x is given, thus missing the specification $por(\bot, tt) = tt$,
- or output some value rightaway, say tt ($\lambda xy.tt$), thus mising the specification $por(\bot, \bot) = \bot$.

For a formal proof that *por* is not sequentially definable, we refer to [33] (syntactic proof), to [22][section 6.1] (model-theoretic proof), and to [5][section 4.5] (via logical relations). As for *sor*, the story is yet different, there are *two* natural sequential programs for it:

$$\lambda xy.\,\texttt{if}\ x = tt \begin{cases} \texttt{then}\,\texttt{if}\ y = \cdots \\ \texttt{else}\,\texttt{if}\ y = \cdots \end{cases} \qquad \lambda xy.\,\texttt{if}\ y = tt \begin{cases} \texttt{then}\,\texttt{if}\ x = \cdots \\ \texttt{else}\,\texttt{if}\ x = \cdots \end{cases}$$

The starting point of the model of sequential algorithms (next section) was to interpret these two programs as different objects *lsor* and *rsor*. Notice finally that there are many more sequential programs computing *lor*, *ror*, or *sor*. Another program for *lor* might e.g. look like

$$\lambda xy.\,\texttt{if}\ x = tt \begin{cases} \texttt{then}\ tt \\ \texttt{else}\,\texttt{if}\ x = tt \begin{cases} \texttt{then}\ tt \\ \texttt{else}\,\texttt{if}\ y = \cdots \end{cases} \end{cases}$$

Such a "stuttering" program is perfectly correct syntactically. Whether this program is interpreted in the model by an object different from the above program for *lor* is the departure point between the model of sequential algorithm on one hand and the more recent games semantics on the other hand. We shall come back to this point in the next section.

Before we close the section, let us give some rationale for the names used in this section. As the reader might have guessed, the prefixes p, l, r, s, ls, rs stand for "parallel", "left", "right", "left strict", and "right strict", respectively.

3 Symmetric algorithms, sequential algorithms

We introduce enough formal notions to give a self-contained introduction to sequential algorithms, presented in the light of a games interpretation [12] (following work of Lamarche [27]). The proofs are omitted, but can be found in [5][section 14.3], except for what regards the coincidence between the two definitions of composition, for which the proof from [14][section 3.6] can easily be adapted.

Definition 3.1 *A* sequential data structure $\mathbf{S} = (C, V, P)$ *is given by two sets C and V of* cells *and* values*, which are assumed disjoint, and by a collection P of non-empty words p of the form:*

$$c_1 v_1 \cdots c_n v_n \text{ or } c_1 v_1 \cdots c_{n-1} v_{n-1} c_n,$$

where $c_i \in C$ and $v_i \in V$ for all i. Thus any $p \in P$ is alternating and starts with a cell. Moreover, it is assumed that P is closed under non-empty prefixes. We call the elements of P positions *of* $\mathbf{S}$. *We call* move *any element of $M = C \cup V$. We use m to denote a move. A position ending with a value is called a* response*, and a position ending with a cell is called a* query*. We use p (or s, or t), q, and r, to range over positions, queries, and responses, respectively. We denote by Q and R the sets of queries and responses, respectively.*

Let us pause here for some comments and perspective. An important step in the semantic account of sequential computing was taken by Berry, who developed the stable model in which the function *por* is excluded. Winskel described this model more concretely in terms of *event structures*, and Girard proposed a simpler form called *coherence spaces*, that led him to the discovery of linear logic [19] (see also [5][chapters 12 and 13]). In event structures or coherence spaces, data are constructed out of elementary pieces, called *events*, or tokens. For example, the pair of booleans (tt, ff) is obtained as the set of two elementary pieces: $(tt, \perp)$ and $(\perp, ff)$. More precisely and technically, the structure $Bool \times Bool$ as a coherence space has four events: $tt.1$, $ff.1$, $tt.2$, and $ff.2$. Then (tt, ff) is the set $\{tt.1, ff.2\}$.

In a sequential data structure (or in a concrete data structure, not defined here) events are further cut in two "halves": a cell and a value, or an opponent's move and a player's move. The structure $Bool \times Bool$ as an sds has two cells $?.1$ and $?.2$ and has four values $tt.1$, $ff.1$, $tt.2$, and $ff.2$. An event, say $tt.1$, is now decomposed as a position $(?.1)\,(tt.1)$. The best way to understand this is to think of a streamlike computation. Your pair of booleans is the output of some program, which will only work on demand. The cell $?.1$ reads as "I – another program, or an observer – want to know the left coordinate of the result of the program", and $tt.1$ is the answer to this query.

An important remark, which will be further exploited in section 5, is that this decomposition of events gives additional space: there is no counterpart in the world of coherence spaces or in any other usual category of domains of a structure with one cell and no value.

Definition 3.2 *A* strategy *of* $\mathbf{S}$ *is a subset* x *of* R *that is closed under response prefixes and binary non-empty greatest lower bounds (glb's):*

$$r_1, r_2 \in x, r_1 \wedge r_2 \neq \epsilon \quad \Rightarrow \quad r_1 \wedge r_2 \in x \ ,$$

where ϵ *denotes the empty word. A* counter-strategy *is a non-empty subset of* Q *that is closed under query prefixes and under binary glb's. We use* $x, y, \ldots$ *and* $\alpha, \beta, \ldots$ *to range over strategies and counter-strategies, respectively.*

If x *is a strategy and if* $r \in x, q = rc$ *for some* c *and if there is no* v *such that* $qv \in x$*, we write* $q \in A(x)$ *(and say that* q *is accessible from* x*). Likewise we define* $r \in A(\alpha)$ *for a response* r *and a counter-strategy* α*.*

Both sets of strategies and of counter-strategies are ordered by inclusion. They are denoted by $D(\mathbf{S})$ *and* $D^{\perp}(\mathbf{S})$*, respectively. We write* $\mathcal{K}(D(\mathbf{S}))$ *and* $\mathcal{K}(D^{\perp}(\mathbf{S}))$ *for the sets of* finite *strategies and counter-strategies, respectively. Notice that* $D(\mathbf{S})$ *has always a minimum element (the empty strategy, written* $\emptyset$ *or* $\perp$*), while* $D^{\perp}(\mathbf{S})$ *has no minimum element in general.*

A more geometric reading of the definitions of sds, strategy and counter-strategy is the following. An sds is a labelled forest, where the ancestor relation alternates cells and values, and where the roots are labelled by cells. A strategy is a sub-forest which is allowed to branch only at values. A counter-strategy α is a non-empty subtree which is allowed to branch only at cells.

Let us see what collections of positions form and do not form a strategy in *Bool* $\times$ *Bool*. The set $\{(?.1)\ (tt.1)\ ,\ (?.2)\ (ff.2\}\}$ (representing (tt, ff)) is a strategy, while $\{(?.1)\ (tt.1)\ ,\ (?.1)\ (ff.1)\}$ is not a strategy. A way to understand this is to say that the cell ?.1 can hold only one value, which is the answer to the question. A strategy consists in having ready determinate answers for the movements of the opponent. If strategies are data, what are counter-strategies? They can be considered as exploration trees, see below.

The pairs cell–value, query–response, and strategy–counter-strategy give to sds's a flavour of symmetry. These pairs are related to other important dualities in programming: input–output, constructor–destructor (see [17]). It is thus tempting to consider the counter-strategies of an sds $\mathbf{S}$ as the strategies of a dual structure $\mathbf{S}^{\perp}$ whose cells are the values of $\mathbf{S}$ and whose values are the cells of $\mathbf{S}$. However, the structure obtained in this way is not an sds anymore, since positions now start with a value. This situation, first analysed by Lamarche [28], is now well-understood since the thesis work of Laurent [29]. We come back to this below.

The following definition resembles quite closely to the dynamics described in section 1.

Definition 3.3 (play) *Let* $\mathbf{S}$ *be an sds,* x *be a strategy and* α *be a counter-strategy of* $\mathbf{S}$*, one of which is finite. We define* $x \mid \alpha$*, called a* play*, as the set of positions* p *which are such that all the response prefixes of* p *are in* x *and all the query prefixes of* p *are in* α*.*

Proposition 3.4 *Given* x *and* α *as in definition 3.3, the play* $x \mid \alpha$ *is non-empty and totally ordered, and can be confused with its maximum element, which is uniquely characterized as follows:*

$$\begin{array}{ll} x \mid \alpha \text{ is the unique element of } x \cap A(\alpha) & \text{if } x \mid \alpha \text{ is a response} \\ x \mid \alpha \text{ is the unique element of } \alpha \cap A(x) & \text{if } x \mid \alpha \text{ is a query .} \end{array}$$

Definition 3.5 (winning) *Let* x *and* α *be as in definition 3.3. If* $x \mid \alpha$ *is a response, we say that* x *wins against* α*, and we denote this predicate by* $x \triangleleft \alpha$*. If* $x \mid \alpha$ *is a query, we say that* α *wins against* x*, and we write* $x \triangleright \alpha$*, thus* $\triangleright$ *is the negation of* $\triangleleft$*. To stress who is the winner, we write:*

$$x \mid \alpha = \begin{cases} x \triangleleft\!\mid \alpha & \text{when } x \text{ wins} \\ x \mid\!\triangleright \alpha & \text{when } \alpha \text{ wins .} \end{cases}$$

The position $x \mid \alpha$ formalizes the interplay between the player with strategy x and the opponent with strategy α. If $x \mid \alpha$ is a response, then the player wins since he made the last move, and if $x \mid \alpha$ is a query, then the opponent wins. Here is a game theoretical reading of $x \mid \alpha$. At the beginning the opponent makes a move c: his strategy determines that move uniquely. Then either the player is unable to move (x contains no position of the form cv), or his strategy determines a unique move. The play goes on until one of x or α does not have the provision to answer its opponent's move (cf. section 1).

We next define the morphisms between sds's. There are two definitions, a concrete one and a more abstract one. The concrete one is needed since we want the morphisms to form in turn an sds in order to get a cartesian closed category (actually a monoidal closed one, to start with). Accordingly, there will be two definitions of the composition of morphisms. Their equivalence is just what *full abstraction* – that is, the coincidence of operational and denotational semantics – boils down to, once we have tailored the model to the syntax (programs as morphisms) and tailored the syntax to the semantics (like in the language CDS [7]). We start with the concrete way.

Definition 3.6 *Given sets* $A, B \subseteq A$*, for any word* $w \in A^*$*, we define* $w\lceil_B$ *as follows:*

$$\epsilon\lceil_B = \epsilon \qquad wm\lceil_B = \begin{cases} w\lceil_B & \text{if } m \in A \backslash B \\ (w\lceil_B)m & \text{if } m \in B \text{ .} \end{cases}$$

Definition 3.7 *Given two sds's* $\mathbf{S} = (C, V, P)$ *and* $\mathbf{S}' = (C', V', P')$, *we define* $\mathbf{S} \multimap \mathbf{S}' = (C'', V'', P'')$ *as follows. The sets* C'' *and* V'' *are disjoint unions:*

$$\begin{array}{rcl} C'' & = & \{\mathit{request}\ c' \mid c' \in C'\} \cup \{\mathit{is}\ v \mid v \in V\} \\ V'' & = & \{\mathit{output}\ v' \mid v' \in V'\} \cup \{\mathit{valof}\ c \mid c \in C\}\ . \end{array}$$

P'' *consists of the alternating positions* s *starting with a request* c', *and which are such that:*

$$\begin{array}{l} s\lceil_{\mathbf{S}'} \in P', (s\lceil_{\mathbf{S}} = \epsilon \text{ or } s\lceil_{\mathbf{S}} \in P), \text{ and} \\ s \text{ has no prefix of the form } s(\mathit{valof}\ c)(\mathit{request}\ c'). \end{array}$$

We often omit the tags request, valof, is, output, as we have just done in the notation $s\lceil_{\mathbf{S}} = s\lceil_{C \cup V}$ *(and similarly for* $s\lceil_{\mathbf{S}'}$*).*

We call affine sequential algorithms *(or affine algorithms) from* $\mathbf{S}$ *to* $\mathbf{S}'$ *the strategies of* $\mathbf{S} \multimap \mathbf{S}'$.

The constraint 'no scc'' can be formulated more informally as follows. Thinking of *valof* c as a call to a subroutine, the principal routine cannot proceed further until it receives a result v from the subroutine.

The identity affine algorithm $id \in D(\mathbf{S} \multimap \mathbf{S}')$ is defined as follows:

$$id = \{\mathit{copycat}(r) \mid r \text{ is a response of } \mathbf{S}\},$$

where *copycat* is defined as follows:

$$\begin{array}{rcl} \mathit{copycat}(\epsilon) & = & \epsilon \\ \mathit{copycat}(rc) & = & \mathit{copycat}(r)(\mathit{request}\ c)(\mathit{valof}\ c) \\ \mathit{copycat}(qv) & = & \mathit{copycat}(q)(\mathit{is}\ v)(\mathit{output}\ v)\ . \end{array}$$

The word *copycat* used in the description of the identity algorithm has been proposed in [1], and corresponds to a game theoretical understanding: the player always repeats the last move of the opponent. In some influential talks, Lafont had taken images from chess (Karpov – Kasparov) to explain the same thing.

Exercise 3.8 (1) *The following affine algorithm computes the boolean negation function:*

$$\begin{array}{l} \{(\mathit{request}\ ?)(\mathit{valof}\ ?), \\ (\mathit{request}\ ?)(\mathit{valof}\ ?)(\mathit{is}\ tt)(\mathit{output}\ ff), \\ (\mathit{request}\ ?)(\mathit{valof}\ ?)(\mathit{is}\ ff)(\mathit{output}\ tt)\}\ . \end{array}$$

(2) *On the other hand, the left disjunction function cannot be computed by an affine algorithm. Indeed, transcribing the program for lor as a strategy leads to:*

$$\begin{array}{l}
\{(\textit{request}\ ?)(\textit{valof}\ ?.1),\\
(\textit{request}\ ?)(\textit{valof}\ ?.1)(\textit{is}\ tt)(\textit{output}\ tt),\\
(\textit{request}\ ?)(\textit{valof}\ ?.1)(\textit{is}\ f\!f)(\textit{valof}\ ?.2),\\
(\textit{request}\ ?)(\textit{valof}\ ?.1)(\textit{is}\ f\!f)(\textit{valof}\ ?.2)(\textit{is}\ tt)(\textit{output}\ tt),\\
(\textit{request}\ ?)(\textit{valof}\ ?.1)(\textit{is}\ f\!f)(\textit{valof}\ ?.2)(\textit{is}\ f\!f)(\textit{output}\ f\!f)\}\ ,
\end{array}$$

which is not a subset of the set of positions of $Bool^2 \multimap Bool$, *because the projections on* $Bool^2$ *of the last two sequences of moves are not positions of* $Bool^2$. *But the program does transcribe into a (non-affine) sequential algorithm, as we shall see.*

(3) Every constant function gives rise to an affine algorithm, whose responses have the form $(\textit{request}\ c'_1)(\textit{output}\ v'_1)\ldots(\textit{request}\ c'_n)(\textit{output}\ v'_n)$..

The second and third example above thus justify the terminology affine (in the affine framework, in contrast to the linear one, weakening is allowed). The second example suggests the difference between affine and general sequential algorithms. Both kinds of algorithms ask successive queries to their input, and continue to proceed only after they get responses to these queries. An affine algorithm is moreover required to ask these queries monotonically: each new query must be an extension of the previous one. The 'unit' of resource consumption is thus a sequence of queries/responses that can be arbitrarily large, as long as it builds a position of the input sds. The disjunction algorithms are not affine, because they may have to ask successively the queries ?.1 and ?.2, which are not related by the prefix ordering.

A generic affine algorithm, as represented in figure 1, can be viewed as a 'combination' of the following (generic) output strategy and input counter-strategy (or exploration tree):

$$\begin{array}{cc}
\text{input counter-strategy} & \text{output strategy}\\
c \left\{ \begin{array}{l} v_1\ \cdots \\ \vdots \\ v_i\ d \left\{ \begin{array}{l} \vdots \\ w \\ \vdots \end{array} \right. \\ \vdots \\ v_n\ \cdots \end{array} \right. &
c'\ v' \left\{ \begin{array}{l} c'_1\ \cdots \\ \vdots \\ c'_m\ \cdots \end{array} \right.
\end{array}$$

$$\textit{request } c' \ \textit{valof } c \left\{ \begin{array}{l} \textit{is } v_1 \ \cdots \\ \vdots \\ \textit{is } v_i \ \textit{valof } d \left\{ \begin{array}{l} \vdots \\ \textit{is } w \ \textit{output } v' \left\{ \begin{array}{l} \textit{request } c'_1 \ \cdots \\ \vdots \\ \textit{request } c'_m \ \cdots \end{array}\right. \\ \vdots \end{array}\right. \\ \vdots \\ \textit{is } v_n \ \cdots \end{array}\right.$$

Figure 1: A generic affine algorithm

We now give a definition of composition of affine algorithms by means of a simple *abstract machine*. Sequential algorithms are syntactic objects, and were indeed turned into a programming language called CDS [7]. What we present here is a simplified version of the operational semantics presented in [14][section 3.5] in the special case of affine algorithms. Given $\phi \in D(\mathbf{S} \multimap \mathbf{S}')$ and $\phi' \in D(\mathbf{S}' \multimap \mathbf{S}'')$, the goal is to compute on demand the positions that belong to their composition ϕ'' in the sds $\mathbf{S} \multimap \mathbf{S}''$. The abstract machine proceeds by rewriting triplets (s, s', s'') where s, s', s'' are positions of $\mathbf{S} \multimap \mathbf{S}'$, $\mathbf{S}' \multimap \mathbf{S}''$, and $\mathbf{S} \multimap \mathbf{S}''$, respectively. The rules are given in Figure 2 (where P'' designates the set of positions of $\mathbf{S} \multimap \mathbf{S}''$, etc...):

The first two rules are left to the (streamlike) initiative of the observer. Each time one of these rules is activated, it launches the machine proper, that consists of the four other (deterministic) rules. The generic behaviour of the machine is as follows. The initial triplet is $(\epsilon, \epsilon, \epsilon)$. The observer wants to know the content of c'', or more precisely wants to know what the function does in order to compute the contents of c'' in the output. Thus, he chooses to perform the following rewriting:

$$(\epsilon, \epsilon, \epsilon) \quad \longrightarrow \quad (\epsilon, \epsilon, c'')\ .$$

The request is transmitted to ϕ':

$$(\epsilon, \epsilon, c'') \quad \longrightarrow \quad (\epsilon, c'', c'')\ .$$

There are two cases here. Either ϕ' does not consult its input and produces immediately a value for c'', in which case, this value is transmitted as the

$$\begin{array}{lcll}
(r,r',r'') & \longrightarrow & (r,r'c'',r''c'') & (r''c''\in P'')\\
(r,r',r'') & \longrightarrow & (rv,r'c'',r''v) & (r''v\in P'')\\
\\
(r,q',q'') & \longrightarrow & (r,q'v'',q''v'') & (q'v''\in\phi')\\
(r,q',q'') & \longrightarrow & (rc',q'c',q'') & (q'c'\in\phi')\\
(q,r',q'') & \longrightarrow & (qv',r'v',q'') & (qv'\in\phi)\\
(q,r',q'') & \longrightarrow & (qc,r',q''c) & (qc\in\phi)
\end{array}$$

Figure 2: Composition abstract machine for affine algorithms

final result of the oberver's query:

$$(\epsilon,c'',c'') \quad\longrightarrow\quad (\epsilon,c''v'',c''v'') \qquad (c''v''\in\phi')\ .$$

Or ϕ' needs to consult its input (like the various sequential or functions), and then an *interaction loop* (in the terminology of Abramsky and Jagadeesan [2]) is initiated:

$$\begin{array}{lcll}
(\epsilon,c'',c'') & \longrightarrow & (c'_1,c''c'_1,c'') & (c''c'_1\in\phi')\\
 & \longrightarrow & (c'_1v'_1,c''c'_1v'_1,c'') & (c'_1v'_1\in\phi)\\
 & \longrightarrow & (c'_1v'_1c'_2,c''c'_1v'_1c'_2,c'') & (c''c'_1v'_1c'_2\in\phi')\\
 & \vdots & &
\end{array}$$

This dialogue between ϕ and ϕ' may terminate in two ways. Either at some stage ϕ' has received enough information from ϕ to produce a value v'' for c'', i.e. $c'_1v'_1\ldots c'_nv'_nv''\in\phi'$:

$$\begin{array}{l}
(c'_1v'_1\ldots c'_nv'_n,c''c'_1v'_1c'_2\ldots c'_nv'_n,c'')\\
\quad\longrightarrow (c'_1v'_1\ldots c'_nv'_n,c''c'_1v'_1c'_2\ldots c'_nv'_nv'',c''v'')
\end{array}$$

or ϕ itself says it needs to consult its input, i.e., $c'_1v'_1\ldots c'_nc\in\phi$: this information is passed as a final (with respect to the query c'') result to the observer, who then knows that ϕ'' needs to know the content of c.

$$\begin{array}{l}
(c'_1v'_1\ldots c'_n,c''c'_1v'_1c'_2\ldots c'_n,c'')\\
\quad\longrightarrow (c'_1v'_1\ldots c'_nc,c''c'_1v'_1c'_2\ldots c'_n,c''c)\ .
\end{array}$$

It is then the observer's freedom to explore further the semantics of ϕ'' by issuing a new query (provided it is in P'') :

$$\begin{array}{l}
(c'_1v'_1\ldots c'_nv'_n,c''c'_1v'_1c'_2\ldots c'_nv'_nv'',c''v'')\\
\quad\longrightarrow (c'_1v'_1\ldots c'_nv'_n,c''c'_1v'_1c'_2\ldots c'_nv'_nv'',c''v''c''_1)
\end{array}$$

or

$$(c'_1 v'_1 \ldots c'_n c, c'' c'_1 v'_1 c'_2 \ldots c'_n, c'' c) \\ \longrightarrow (c'_1 v'_1 \ldots c'_n c, c'' c'_1 v'_1 c'_2 \ldots c'_n, c'' cv)\ .$$

The query $c''cv$ reads as: "knowing that ϕ'' needs c, how does it behave next when I feed v to c"? After this, the computation starts again using the four deterministic rules along the same general pattern. Notice how ϕ and ϕ' take the leadership in turn (cf. section 1).

We now turn to the abstract definition of our morphisms.

Definition 3.9 *A (continuous) function $f : D(\mathbf{S}) \to D(\mathbf{S}')$ is called* stable *if for any $x \in D(\mathbf{S}), \alpha' \in \mathcal{K}(D^\perp$ such that $\mathbf{S}')), f(x) \lhd \alpha'$ there exists a minimum (finite) $y \le x$ such that $f(y) \lhd \alpha'$ ($m(g, \alpha', x)$, denoted by $m(f, x, \alpha')$. One defines similarly a notion of stable function $g : D^\perp(\mathbf{S}') \rightharpoonup D^\perp(\mathbf{S})$, with notation $m(g, \alpha', x)$.*

Definition 3.10 (symmetric algorithm) *Let $\mathbf{S}$ and $\mathbf{S}'$ be two sds's. A symmetric algorithm from $\mathbf{S}$ to $\mathbf{S}'$ is a pair*

$$(f : D(\mathbf{S}) \to D(\mathbf{S}'), g : D^\perp(\mathbf{S}') \rightharpoonup D^\perp(\mathbf{S}))$$

of a function and a partial function that are both continuous and satisfy the following axioms:

$$\text{(L)}\ \ (x \in D(\mathbf{S}), \alpha' \in \mathcal{K}(D^\perp(\mathbf{S}')), f(x) \lhd \alpha') \Rightarrow \begin{cases} x \lhd g(\alpha') \text{ and} \\ m(f, x, \alpha') = x \lhd\!\mid g(\alpha') \end{cases}$$

$$\text{(R)}\ \ (\alpha' \in D^\perp(\mathbf{S}'), x \in \mathcal{K}(D(\mathbf{S})), x \rhd g(\alpha') \Rightarrow \begin{cases} f(x) \rhd \alpha' \text{ and} \\ m(g, \alpha', x) = f(x) \mid\!\rhd \alpha' \end{cases}$$

We set as a convention, for any x and any α' such that $g(\alpha')$ is undefined:

$$x \lhd g(\alpha') \text{ and } x \lhd\!\mid g(\alpha') = \emptyset.$$

Hence the conclusion of (L) is simply $m(f, x, \alpha') = \emptyset$ when $g(\alpha')$ is undefined. In contrast, when we write $x \rhd g(\alpha')$ in (R), we assume that $g(\alpha')$ is defined.

Thus, g provides the witnesses of stability of f, and conversely. Moreover, the above definition is powerful enough to imply other key properties of f and g.

Definition 3.11 *A (continuous) function* $f : D(\mathbf{S}) \to D(\mathbf{S}')$ *is called* sequential *if, for any pair* $(x, \alpha') \in \mathcal{K}(D(\mathbf{S})) \times \mathcal{K}(D^{\perp}(\mathbf{S}'))$ *such that* $f(x) \rhd \alpha'$ *and* $f(z) \lhd \alpha'$ *for some* $z \geq x$*, there exists* $\alpha \in \mathcal{K}(D^{\perp}(\mathbf{S}))$*, called a* sequentiality index *of* f *at* (x, α')*, such that* $x \rhd \alpha$ *and for any* $y \geq x$*,* $f(y) \lhd \alpha'$ *implies* $y \lhd \alpha$*.*

Proposition 3.12 *Let* f *and* g *be as in the previous definition. Then* f *and* g *are* affine *(i.e., preserve the least upper bounds of pairs of upper bounded elements) and satisfy the following two axioms:*

(LS) If $x \in D(\mathbf{S})$*,* $\alpha' \in \mathcal{K}(D^{\perp}(\mathbf{S}'))$*,* $f(x) \rhd \alpha'$*, and* $f(y) \lhd \alpha'$ *for some* $y > x$*, then* $x \rhd g(\alpha')$*, and* $x \mid\!\rhd\, g(\alpha')$ *is a sequentiality index of* f *at* (x, α')*.*

(RS) If $\alpha' \in D^{\perp}(\mathbf{S}')$*,* $x \in \mathcal{K}(D(\mathbf{S}))$*,* $x \lhd g(\alpha')$*, and* $x \rhd g(\beta')$ *for some* $\beta' > \alpha'$*, then* $f(x) \lhd \alpha'$*, and* $f(x) \lhd\!\mid \alpha'$ *is a sequentiality index of* g *at* (α', x)*. Hence* f *and* g *are sequential, and* g *provides the witnesses of sequentiality for* f *and conversely.*

We turn to the composition of affine algorithms.

Definition 3.13 *Let* $\mathbf{S}$*,* $\mathbf{S}'$ *and* $\mathbf{S}''$ *be sds's, and let* (f, g) *and* (f', g') *be symmetric algorithms from* $\mathbf{S}$ *to* $\mathbf{S}'$ *and from* $\mathbf{S}'$ *to* $\mathbf{S}''$*. We define their composition* (f'', g'') *from* $\mathbf{S}$ *to* $\mathbf{S}''$ *as follows:*

$$f'' = f' \circ f \quad \text{and} \quad g'' = g \circ g'.$$

The announced full abstraction theorem is the following.

Theorem 3.14 *The sets of affine algorithms and of symmetric algorithms are in a bijective correspondence (actually, an isomorphism), and the two definitions of composition coincide up to the correspondence.*

We just briefly indicate how to pass from one point of view to the other. Given $\phi \in D(\mathbf{S} \multimap \mathbf{S}')$, we define a pair (f, g) of a function and a partial function as follows:

$$\begin{array}{rcl} f(x) & = & \{r' \mid r' = s\lceil_{\mathbf{S}'} \text{ and } s\lceil_{\mathbf{S}} \in x \text{ for some } s \in \phi\} \\ g(\alpha') & = & \{q \mid q = s\lceil_{\mathbf{S}} \text{ and } s\lceil_{\mathbf{S}'} \in \alpha' \text{ for some } s \in \phi\}. \end{array}$$

(By convention, if the right hand side of the definition of g is empty for some α', we interpret this definitional equality as saying that $g(\alpha')$ is undefined.)

Conversely, given a symmetric algorithm (f, g) from $\mathbf{S}$ to $\mathbf{S}'$, we construct an affine algorithm $\phi \in D(\mathbf{S} \multimap \mathbf{S}')$ by building the positions s of ϕ by induction on the length of s (a streamlike process!). For example, if $s \in \phi$, if $s\lceil_{\mathbf{S}}$ and $s\lceil_{\mathbf{S}'}$ are responses, and if $q' = (s\lceil_{\mathbf{S}'})c'$ for some c', then:

$$\begin{array}{ll} sc'c \in \phi & \text{if } (s\lceil_{\mathbf{S}})c \in g(q') \\ sc'v' \in \phi & \text{if } q'v' \in f(s\lceil_{\mathbf{S}}) \ . \end{array}$$

But, as remarked above, we do not get all sequential functions in this way. Recall that in linear logic the usual implication $A \Rightarrow B$ is decomposed as $(!A) \multimap B$ (!, and its de Morgan dual ?, are called *exponentials* in linear logic).

Definition 3.15 (exponential) *Let* $\mathbf{S} = (C, V, P)$ *be an sds. We set* $!\mathbf{S} = (Q, R, P_!)$, *where* Q *and* R *are the sets of queries and of responses of* $\mathbf{S}$, *respectively, and where* $P_!$ *is recursively specified as follows (letting* ρ *range over responses in* $P_!$*):*

$$\begin{array}{ll} \rho q \in P_! & \textit{if } q \in A(\texttt{strategy}(\rho)) \\[1em] \rho q(qv) \in P_! & \textit{if } \rho q \in P_!, \texttt{strategy}(\rho q(qv)) \in D(M), \\ & \textit{and } qv \notin \texttt{strategy}(\rho) \ , \end{array}$$

where `strategy` *is the following function mapping responses (or* ϵ*) of* $P_!$ *to strategies of* $\mathbf{S}$*:*

$$\texttt{strategy}(\epsilon) = \emptyset \qquad \texttt{strategy}(\rho q(qv)) = \texttt{strategy}(r) \cup \{qv\}.$$

Sequential algorithms *between two sds's* $\mathbf{S}$ *and* $\mathbf{S}'$ *are by definition affine algorithms between* $!\mathbf{S}$ *and* $\mathbf{S}'$.

It is easily checked that the programs for *lor* (cf. example 3.8), *ror*, *lsor*, and *rsor* transcribe as sequential algorithms from $Bool \times Bool$ to $Bool$.

Sequential algorithms also enjoy two direct definitions, a concrete one and an abstract one, and both an operational and a denotational definition of composition, for which full abstraction holds, see [14].

Let us end the section with a criticism of the terminology of symmetric algorithm. As already pointed out, the pairs (f, g) are not quite symmetric since g unlike f is a *partial* function. Logically, $\mathbf{S} \multimap \mathbf{S}'$ should read as $\mathbf{S}^\perp \parr \mathbf{S}'$. But something odd is going on: the connective $\parr$ would have two arguments of a different *polarity*: in $\mathbf{S}'$ it is Opponent who starts, while Player starts in $\mathbf{S}^\perp$. For this reason, Laurent proposed to decompose the affine arrow [29] (see also [8]):

$$\mathbf{S} \multimap \mathbf{S}' = (\downarrow \mathbf{S})^\perp \parr \mathbf{S}' \ ,$$

where $\downarrow$ is a change of polarity operator. For sds's, this operation is easy to define: add a new initial opponent move, call it $\star$, and prefix it to all the positions of $\mathbf{S}^{\perp}$. For example, $\downarrow (Bool^{\perp})$ has $\star\ ?\ tt$ and $\star\ ?\ ff$ as (maximal) positions. According to Laurent's definition, the initial moves of $\mathbf{S_1} \parr \mathbf{S_2}$ are pairs (c_1, c_2) of initial (Opponent's) moves of $\mathbf{S_1}$ and $\mathbf{S_2}$. Then the positions continue as interleavings of a position of $\mathbf{S_1}$ and of $\mathbf{S_2}$. Notice that this is now completely symmetric in $\mathbf{S_1}$ and $\mathbf{S_2}$.

Now, let us revisit the definition of $\mathbf{S} \multimap \mathbf{S}'$. We said that the positions of this sds had to start with a c', which is quite dissymetric. But the $\downarrow$ construction allows us to restore equal status to the two components of the $\parr$. A position in $\mathbf{S}^{\perp} \parr \mathbf{S}'$ must start with two moves played together in $\mathbf{S}$ and $\mathbf{S}'$. It happens that these moves have necessarily the form $(\star, c')$, which is conveying the same information as c'.

4 Related works

Sequential algorithms turned out to be quite central in the study of sequentiality. First, let us mention that Kleene has developed (for lower types) similar notions [25], under the nice name of oracles, in his late works on the semantics of higher order recursion theory (see [9] for a detailed comparison).

Two important models of functions that have been constructed since turned out to be the extensional collapse (i.e. the hereditary quotient equating sequential algorithms computing the same function, i.e. (in the affine case) two algorithms (f, g) and (f', g') such that $f = f'$): Bucciarelli and Ehrhard's model of strongly stable functions [10, **?**], and Longley's model of sequentially realizable functionals [31]. The first model arose from an algebraic characterization of sequential (first-order) functions, that carries over to all types. The second one is a realizability model over a combinatory algebra in which the interaction at work in sequential algorithms is encoded.

Also, Laird has shown that sequential algorithms can be obtained by a collapsing construction from his games model of control in Hyland and Ong style [26].

The models of Hyland and Ong and of Abramsky, Jagadeesan and Malacaria (HO and AJM, respectively) capture PCF definability exactly, whereas the games associated with sequential algorithms also accommodate control operations such as `call-cc` that are not definable in PCF (see section 5). In fact, the interpretation function from normal forms to these models is injective. An essential merit of these works was to characterize the image of this injection, and hence to characterize PCF definability in a syntax-independent way by a few conditions such as innocence and well-bracketing. This opened the way to a whole research program launched by Abramsky. What does happen if one of the conditions is relaxed? Giving up innocence led to very interesting (and fully abstract) models of references (see [4]). Giving up well-bracketing gave a model of PCF plus control, as already mentioned.

The model of sequential algorithms and the HO (or AJM) model differ drastically in size. The type $Bool \to Bool$ is interpreted by a finite sds (i.e., an sds with finitely many positions) in the model of sequential algorithms, while there are infinitely many PCF Böhm trees (and hence infinitely many strategies) in the HO and AJM models at that type. The difference comes from the way the exponential is defined. In definition 3.15, a key feature is non-repetition ($qv \notin \mathtt{strategy}(\rho)$). In the games models, the exponential is defined either by interleaving allowing for repetitions or by the opening of potentially infinitely many copies of positions. Roughly, this amounts to dropping the condition $qv \notin \mathtt{strategy}(\rho)$.

The finitary nature of sequential algorithms implies that equality in the model is decidable for any type built over $Bool$, while the term model games do not provide effective tools to tackle observational equivalences. As a matter of fact, it has been proved by Loader [30] that equality of two objects in the fully abstract model of (finitary) PCF is undecidable. A model of PCF is called fully abstract if it equates two terms if and only if these terms are *observationally equivalent*, which means that one can be replaced by the other in any program context without affecting the final result. The *full abstraction problem* of Scott, Milner and Plotkin was the (quite loosely) specified problem of providing a *denotational* construction of the fully abstract model of PCF, as opposed to the original term-based construction of Milner, who also had shown the uniqueness of the fully abstract model [32]. The HO and AJM games models can be called denotational, since they provide a syntax-independent characterization of a term model made of (a PCF version of) Böhm trees. But they yield full abstraction only via a collapse construction which is not essentially different from the one originally performed by Milner. An implicit hope was to arrive at decidability results for the equality in the model, as usual denotational models consist of functions, and hence interpret every type built over Bool by a finite set. Loader's result says that there cannot be such a construction of the fully abstract model of PCF, and justifies a posteriori why game models had to use infinitary exponentials. In contrast, when PCF is extended with control, then the finitary exponential of the model of sequential algorithms does the job (coming next).

5 Control

We already pointed out that theorem 3.14 is a full abstraction result (for the affine case), and that the same theorem has been proved for all sequential algorithms with respect to the language CDS. Sequential algorithms allow inherently to consult the internal behaviour of their arguments and to make decisions according to that behaviour. For example, there exists a sequential algorithm of type $(Bool^2 \to Bool) \to Bool$ that maps *lsor* to *tt* and *rsor* to *ff* (cf. end of section 2). Cartwright and Felleisen made the connection with more standard control operators explicit, and this lead to the full abstraction result of sequential algorithms with respect to an extension of PCF with a control operator [13].

In this respect, the following is a key observation (see [26]). Let o be the sds with one cell and no value: $o = \{\{?\}, \emptyset, \{?\}\}$. Then we have the isomorphism

$$Bool \sim (o \to o \to o) \ ,$$

where $Bool$ is the sds $\{\{?\}, \{tt, ff\}, \{?, (?\ tt), (?\ ff)\}\}$ considered above. Indeed, both sds's have exactly three strategies, ordered in the same way:

$$\begin{array}{l} D(Bool) = \{\emptyset, \{?\ tt\}, \{?\ ff\}\} \\ D(o \to o \to o) = \{\emptyset, \{?_\epsilon\ ?_1\}, \{?_\epsilon\ ?_2\}\} \end{array}$$

(we use subscripts to decorate the cells of the three copies of o, using the convention $o_1 \to o_2 \to o_\epsilon$). It is an instructive exercise to write down explicitly the inverse isomorphisms as sequential algorithms: in one direction, one has the `if then else` function, in the other direction, we have the control operation `catch` considered in [13], which tells apart the two strategies $, \{?_\epsilon\ ?_1\}, \{?_\epsilon\ ?_2\}$. Here, we shall show (at type *bool*) how the control operator `call-cc` of Scheme or Standard ML is interpreted as a sequential algorithm of type $((bool \to B) \to bool) \to bool$. The formula $((A \to B) \to A) \to A$ is called Pierce's law and is a typical tautology of classical logic. The connection between control operators and classical logic – and in particular the fact that `call-cc` corresponds to Pierce's law– was first discovered in [21]. Here is is the sequential algorithm interpreting `call-cc` for $A = bool$:

$$?_\epsilon\ ?_1 \left\{ \begin{array}{l} ?_{11}\ ?_{111} \left\{ \begin{array}{l} tt_{111}\ tt_\epsilon \\ ff_{111}\ ff_\epsilon \end{array} \right. \\ tt_1\ tt_\epsilon \\ ff_1\ ff_\epsilon \end{array} \right.$$

(with labelling of moves $((bool_{111} \rightarrow B_{11}) \rightarrow bool_1) \rightarrow bool_\epsilon)$. The same algorithm, with *bool* replaced by $o \rightarrow o \rightarrow o$, is:

$$?_\epsilon\ ?_1 \begin{cases} ?_{11}\ ?_{111} \begin{cases} ?_{1111}\ ?_2 \\ ?_{1112}\ ?_3 \end{cases} \\ ?_{12}\ ?_2 \\ ?_{13}\ ?_3 \end{cases}$$

(with labelling $(((o_{1111} \rightarrow o_{1112}) \rightarrow o_{111} \rightarrow B_{11}) \rightarrow o_{12} \rightarrow_{13}\rightarrow o_1) \rightarrow o_2 \rightarrow o_3 \rightarrow o_\epsilon)$. The reader familiar with continuations may want to compare this tree with the continuation-passing (CPS) style interpretation $\lambda yk.y(\lambda xk'.xk)k$ of `call-cc`, or in tree form (cf. section 1):

$$\lambda yk.y \begin{cases} \lambda xk'.x \{\ k \\ k \end{cases}$$

(where the first k indicates a copy-cat from o_{111} to o_ϵ while the second one indicates a copycat from o_1 to o_ϵ). This is an instance of the injection from terms to strategies mentioned in section 4 (in this simple example, Laird's HO style model coincides with that of sequential algorithms).

CPS translations are the usual indirect way to interpret control operators: first translate, then interpret in your favorite cartesian closed category. In contrast, sequential algorithms look as a direct semantics. The example above suggests that this is an "illusion": once we explicitly replace *bool* by $o \rightarrow o \rightarrow o$, we find the indirect way underneath.

A more mathematical way to stress this is through thej notion of continuation model [23]: given a category having all the function spaces $A \rightarrow R$ for some fixed object R called object of final results, one only retains the full subcategory of *negative* objects, that is, objects of the form $A \rightarrow R$. In this category, control can be interpreted. (For the logically inclined reader, notice that thinking of R as the formula "false", then the double negation of A reads as $(A \rightarrow R) \rightarrow R$, and the classical tautology $((A \rightarrow R) \rightarrow R) \rightarrow A$ is intuitionistically provable for all negative $A = B \rightarrow R$.) Now, taking $R = o$, the above isomorphism exhibits *bool* as a negative object. But then all types are negative: given A and $B = B' \rightarrow R$, then $A \rightarrow B \sim (A \times B') \rightarrow R$ is also negative. Hence the model of sequential algorithms (and Laird's model of control) are indeed continuation models, but it is not written on their face.

6 A few more remarks

We would like to mention that this whole line of research on sequential interaction induced such side effects as the design of the Categorical Abstract

Machine [11], that gave its name to the language CAML, and of a theory of Abstract Böhm Trees, alluded to in section 1.

As for future lines of research, imports from and into the program of ludics newly proposed by Girard [20] are expected. We just quote one connection with ludics. We insisted in section 2 that *lsor* and *rsor* were different programs for the same function. But there is a way to make them into two different functions, by means of additional *error* values, and accordingly of additional constants in the syntax. Actually, one error is enough, call it *err*. Indeed, we have:

$$lsor(err, \bot) = err \qquad rsor(err, \bot) = \bot \ .$$

Because *lsor* looks at its left argument first, if an error is fed in that argument, it is propagated, whence the result *err*. Because *rsor* looks at its right argument first, if no value is is fed for that argument, then the whole computation is waiting, whence the result $\bot$. One could achieve the same more symmetrically with two different errors: $lsor(err_1, err_2) = err_1$, $rsor(err_1, err_2) = err_2$. But the economy of having just one error is conceptually important, all the more because in view of the isomorphism of section 5, we see that we can dispense (at least for *bool* but also for any finite base type) with the basic values $tt, \mathit{ff}, 0, 1, \ldots$. We arrive then at a picture with only two (base type) constants: $\bot$ and *err*! This is the point of view adopted in Girard's ludics. In ludics, the counterpart of *err* is called Daimon. The motivation for introducing Daimon is quite parallel to that of having errors. Girard's program has the ambition of giving an interactive account of proofs. So, in order to explore a proof of a proposition A, one should play it against a "proof" of $A^\perp$ (the negation of linear logic). But it can't be a proof, since not both A and $A^\perp$ can be proved. So, the space of "proofs" must be enlarged to allow for more opponents to interact with. Similarly, above, we motivated errors by the remark that, once introduced, they allow more observations to be made: here, they allowed us to separate *lsor* and *rsor*. More information, also of a survey kind, can be found in [17].

References

[1] S. Abramsky and R. Jagadeesan, Games and Full Completeness for Multiplicative Linear Logic, Proc. FST-TCS, Springer Lect. Notes in Comp. Sci. 652 (1992).

[2] S. Abramsky and R. Jagadeesan, New foundations for the geometry of iinteraction, Information and Computation 111 (1), 53-119 (1994).

[3] S. Abramsky, R. Jagadeesan, and P. Malacaria, Full abstraction for PCF (1995).

[4] S. Abramsky and G. McCusker, Game semantics, in Computational Logic, U. Berger and H. Schwichtenberg eds, Springer-Verlag, 1-56 (1999).

[5] R. Amadio and P.-L. Curien, Domains and lambda-calculi, Cambridge University Press (1998).

[6] G. Berry and P.-L. Curien, Sequential algorithms on concrete data structures, Theoretical Computer Science 20, 265-321 (1982).

[7] G. Berry and P.-L. Curien, Theory and practice of sequential algorithms: the kernel of the applicative language CDS, in *Algebraic methods in semantics*, Nivat and Reynolds eds., Cambridge University Press, 35-87 (1985).

[8] P. Boudes, PhD thesis (forthcoming).

[9] A. Bucciarelli, Another approach to sequentiality: Kleene's unimonotone functions, Proc. Mathematical Foundations of Programming Semantics, Springer Lect. Notes in Comp. Sci. 802 (1993).

[10] A. Bucciarelli and T. Ehrhard A. Bucciarelli and T. Ehrhard, Sequentiality in an extensional framework, Information and Computation 110(2), 265-296 (1994).

[11] G. Cousineau, P.-L. Curien, and M. Mauny, The categorical abstract machine, Science of Computer Programming 8, 173-202 (1987).

[12] P.-L. Curien, On the symmetry of sequentiality, Proc. Mathematical Foundations of Programming Semantics 1993, Springer Lect. Notes in Comp. Science. 802, 122-130 (1993).

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

[14] P.-L. Curien, Categorical combinators, sequential algorithms and functional programming,Pitman (1986). Revised edition, Birkhäuser (1993).

[15] P.-L. Curien, Abstract Böhm trees, Mathematical Structures in Computer Science 8, 559-591 (1998).

[16] P.-L. Curien and H. Herbelin, Computing with abstract Böhm trees, Third Fuji International Symposium on Functional and Logic Programming, April 1998, Kyoto, Eds Masahiko Sato & Yoshihito Toyama, World Scientific (Singapore), pp 20-39 (1998).

[17] P.-L. Curien, Symmetry and interactivity in programming, to appear in Bulletin of Symbolic Logic.

[18] T. Ehrhard, A relative definability result for strongly stable functions and some corollaries, Information and Computation 152, 111-137 (1999).

[19] J.-Y. Girard, Linear logic, Theoretical Computer Science 50, 1-102 (1987).

[20] J.-Y. Girard, Locus Solum, Mathematical Structures in Computer Science 11(3), 301-506 (2001).

[21] T. Griffin, A formulae-as-types notion of control, Proc. Principles of Programming Languages 1990, ACM Press (1990).

[22] C. Gunter, Semantics of programming languages: structures and techniques, MIT Press (1992).

[23] M. Hofmann and T. Streicher, Continuation models are universal for $\lambda\mu$-calculus, Proc. Logic in Computer Science 1997, Warsaw, IEEE Press (1997).

[24] M. Hyland and L. Ong, On full abstraction for PCF, Information and Computation 163(2), 285-408 (2000).

[25] S. Kleene, Recursive Functionals and Quantifiers of Finite Types Revisited I, II, III, and IV, respectively in Proc. General Recursion Theory II, Fenstad et al. (eds.), North-Holland (1978), Proc. The Kleene Symposium, Barwise et al. (eds.), North-Holland (1980), Proc. Patras Logic Symposium, North Holland (1982), and Proc. Symposia in Pure Mathematics 42 (1985).

[26] J. Laird, Games and sequential algorithms, to appear in Theoretical Computer Science.

[27] F. Lamarche, Sequentiality, games and linear logic, manuscript (1992).

[28] F. Lamarche, Games semantics for full propositional linear logic, Proc. Logic In Computer Science '95, IEEE Computer Society Press (1995).

[29] O. Laurent, Etude de la polarisation en logique, Thèse de Doctorat, Université Aix-Marseille II, march 2002.

[30] R. Loader, Finitary PCF is undecidable, manuscript, University of Oxford (1996), to appear in Theoretical Computer Science.

[31] J. Longley, The sequentially realizable functionals, to appear in Annals of Pure and Applied Logic.

[32] R. Milner, Fully abstract models of typed lambda-calculi, Theoretical Computer Science 4, 1-23 (1977).

[33] G. Plotkin, LCF as a programming language, Theoretical Computer Science 5, 223-257 (1977).