Notes on game semantics

Pierre-Louis Curien (CNRS – Université Paris 7)[*]

February 19, 2019

Abstract

The subject called game semantics grew out as a coherent body of work from two seminal works of the early 1990's, with forerunners in logic, recursion theory, and semantics. Game semantics allows us to provide precise and also natural, interactive semantics to most of the classical features of programming such as functions, control, references. The precision is measured by definability and in some cases by full abstraction results. Applications of game semantics to model-checking and abstract interpretation are being developed, which opens the way for connecting the uses of games in semantics and in verification.

1 Introduction

Semantics is the area of computer science concerned with specifying the meaning of programming constructs. Besides being a field of its own interest, which has developed its own mathematical tools, semantics can help to control the complexity of software architectures. It may guide the design of new languages or of implementation techniques.

Game semantics allows to provide precise and also natural, interactive semantics to most of the classical features of programming such as functions, control, references. It has the flavour of both denotational and operational semantics: mathematical elegance on one hand, and step-by-step modelling of computation on the other hand.

The precision of game semantics is measured by definability and in some cases full abstraction results. A definability result states that each element of the model (or each of the elements of the model of a certain kind) is the interpretation of a term of the language under study. Game semantics has been very successful in establishing such results for various languages, in a structured way. Richer languages correspond to less constrained sorts of game models, and some elements of the less constrained models play a "generic"

[*]Laboratoire *Preuves, Programmes et Systèmes*, Case 7014, 2 place Jussieu, 75251 Paris Cedex 05, France, curien@pps.jussieu.fr.

role in that every element of the less constrained model can be decomposed into an element of the more constrained model and such a generic element.

Full abstraction is the property that a model equates exactly the terms of the language that cannot be distinguished by means of syntactic observations. Full abstraction may be obtained through some abstract construction from definability, by quotienting the model by an observational equivalence that mirrors the syntactic observational equivalence [34]. But in most cases, the model thus obtained does not enjoy a nice direct description.

Game semantics grew out as a coherent body of work from seminal works of the early 1990's, by Abramsky, Jagadeesan and Malacaria on one hand [4], and by Hyland and Ong [61] (as well as by Nickau [90]) on the other hand. The two teams presented each a model of PCF (a paradigmatic core purely functional language) in which types are interpreted by games and terms by strategies. The two models, now commonly referred to as the AJM and the HO model, respectively, have provided the first syntax-independent models of PCF to enjoy the property of definability (for all compact elements of the model). In both cases, the construction of the model of PCF was preceded by the construction of a game model for multiplicative linear logic, with a corresponding definability result [3, 60].

Game semantics has had forerunners in logic, recursion theory, and semantics.

- In logic, the earliest forerunners are Lorenzen and his coworkers [41]. They considered the activity of proving a formula as a strategy in what they caled a *dialogue game* between two players: Proponent, who is responsible for building the proof, and Opponent, who chooses to refute the formula or any of the intermediate conclusions in the (unravelling of the) proof. Unfortunately, most of this work focused on provability more than on proofs themselves.

- In semantics, in the search of a semantic counterpart of sequentiality, Berry and Curien [13] built around 1978 the first interactive model of PCF (and then of a language inspired by the model itself, called CDS [14]), called the model of sequential algorithms on Kahn and Plotkin's concrete data structures [65].

- At around the same time, and completely independently, in a series of works revisiting notions of higher-order computability, Kleene [67, 68, 69, 70] came up with very much the same kind of objects as Berry and Curien (see [18] for a precise comparison).

A more implicit, but nevertheless important reference for the rising of game semantics is the program of the geometry of interaction [46, 47, 49, 50] carried out by Girard in the framework of linear logic [45]. There, much like in the framework of sequential algorithms, one can describe execution in terms of information flow in a fixed network representing a program or a proof.

The most immediate forerunners of game semantics are (in our opinion) the following works:

- Blass games [15] and their recasting in terms of linear logic [16]. The analysis of the limitations of this model (in which composition is not associative in general) has been since an important source of thinking [11, 8].
- Coquand's interaction sequences [24], which provided a first instance of a game semantical account of cut-elimination in a classical setting.
- Lamarche's analysis of the model of sequential algorithms [75, 28] in terms of games.
- Gandy's (unpublished) investigations (with his student Pani) towards the characterization of the dialogues arising from Kleene's work (cf. above).

One should add to this background the fascinating work of Conway [22] which is a reservoir of clever constructions on games, and Joyal's early recognition that games and strategies can be organized in a category [63].

As far as effective full abstraction results are concerned, let us mention that

- Cartwright, Curien and Felleisen have shown that the model of sequential algorithms is fully abstract for an extension of PCF with control operators
- Abramsky and McCusker have tuned HO games to get a fully abstract models of a functional language with imperative features, called Idealized Algol.

In section 2, we introduce game semantics (à la Hyand-Ong) through examples. Section 3 provides some background on the language PCF, as well as on definability and full abstraction. In section 4, we give formal definitions and build categories of games, or arenas, and strategies. Section 5 is devoted to the central definability result of game semantics, which characterizes the (injective) image of PCF Böhm trees (which are suitable possibly infinite normal forms of PCF terms) in the HO model. In section 6, we present the fully abstract game semantics of commands and references. In section 7, we present some applications of game semantics to model-checking and abstract interpretation. In the final section 8, we discuss game semantics in the context of the semantics of proofs.

These notes grew up initially from lectures given at Shanghai Jiaotong University in the summer of 2004, and, jointly with Claudia Faggian, at the university of Padova in the winter of 2006. The present version has been prepared for the Chambéry-Torino summer school in Torino, end of August 2006. I'd like to thank the respective institutions for their hospitality, and the audiences for their reactivity.

2 Programs as strategies

We introduce game semantics via examples. We present various familiar data types as games. The games have two players, called Opponent and Player. Player stands for the program, and Opponent for its environment (the rest of the program, the libraries used by the program, the person experimenting with the program, etc...). A game, or *arena*, is defined by means of a set of moves, each being either a move for Opponent (or O move) or a move for Player (or P move). As we shall see later, the arena also specifies an enabling relation among moves. Some moves are initial, others can be played only if some previous move has been played. Think of the black and the white pieces at chess or draughts! Just as in those games, the players play in turns, and thus create a sequence of moves $m_1 \dots m_n$. We call such a sequence a *play*.

In our games, except in section 8, it is always Opponent who starts, thus $m_1, \dots, m_{2k+1}, \dots$ are O moves while $m_2, \dots, m_{2k}, \dots$ are P moves. The (even-length) plays are elementary pieces out of which we can build data. There are some rules on how to assemble plays to form *strategies*. A good chess player applies some strategy: while the Opponent prepares his next move, Player may think of what to do if Opponent will play this (say, m), and what to do instead if Opponent will play that (say, m'). Hence Player's strategy will typically include plays umn and $um'n'$, where u records whatever was played before. By convention, a strategy is never empty: it contains the empty play ϵ (that is, the unique sequence of moves of length 0). It is also closed under even prefixes. As a matter of fact, the elementary properties of strategies (that they form a category closed under certain constructions) can be proved based on just these two conditions. Then, further conditions can be added, like determinism, or innocence. We shall touch these notions while we develop the examples. How typical data can be interpreted as strategies will become clear through the examples below.

We start with the simplest kind of data type. The arena `nat` for natural numbers has the following moves: one (initial) O move denoted q, and a P move n for each natural number n. All even-length plays have thus the form

$$qn_1qn_2 \dots qn_k$$

A strategy consisting of only the empty sequence corresponds to an undefined value. It is the $\perp$ of Scott denotational semantics [9]. The presence of the play qn corresponds to the natural number n. Both $\{\epsilon, qn\}$ and $\{\epsilon, qn, qnqn, qnqnqn, \dots\}$ seem sensible ways to represent n as a strategy. We shall come back to this later, but for the time being we shall use an anthropomorphic terminology and call $\{\epsilon, qn\}$ and $\{\epsilon, qn, qnqn, qnqnqn, \dots\}$ the *meager* and the *fat* representation of n, respectively. There is a closer match between the meager representation and syntax, but the fat representation is easier to manipulate mathematically. The official point of view of game semantics is the fat one. But for *innocent* strategies (see below), the fat and the meager representations are in a bijective correspondence. For the rest of this introduction, we shall prefer to focus on meager representations, which are easier to interpret.

What about a strategy containing plays such as qn_1qn_2, or a strategy containing both

qn_1 and qn_2 (for $n_1 \neq n_2$)? Such strategies will be considered if we want to give a meaning to a nondeterministic program of type `nat`, whose result may be n_1 or n_2 depending on how the execution went. In these notes, we concentrate on *deterministic* strategies, which are strategies σ subject to the following condition: if $u_1, u_2 \in \sigma$, then their longest common prefix is of even length. This is equivalent to requiring that whenever $umn_1, umn_2 \in \sigma$, then $n_1 = n_2$ (determinism!). To see why these conditions are equivalent, just notice that the negation of the first condition amounts to the existence of two sequences u and v in σ of the form umn_1u' and vmn_2v', where u and v are of even length and $n_1 \neq n_2$ (and hence also $umn_1, vmn_2 \in \sigma$, since strategies are closed under even prefixes). Determinism thus excludes strategies containing both qn_1 and qn_2.

But what about qn_1qn_2? This play will be precluded by the further condition of innocence. In an innocent strategy, Player plays based only on a partial view of the computation, called the *P view* (at the current position). The P view of a play is obtained by removing certain moves preceding O moves. If the O move is initial, then one removes all the moves preceding it. It if it is not initial, then we remove all the moves that stand (strictly) between the O move and its justifier. We will define views more formally later, but for the time being let us just mention that some O moves are initial, and that the player's view of a play sm, where m is initial, is m. Thus the P view of qn_1q is just q, and hence Player's move n_2 must coincide with the move n_1 which he played after q. The meager form of an innocent strategy precisely consists of views only (i.e., of plays whose which are their own view). For the rest of this section, we limit ourselves to innocent strategies in meager form, i.e., presented as sets of views.

But in section 6 we shall relax the innocence constraint and we shall see how to give meaning to sequences like qn_1qn_2, which make sense if one thinks not of a number, but of a memory cell holding a number and which may be overwritten.

In summary, for type `nat` we end up with the following (complete list of) strategies:

$$\begin{cases} \{\epsilon\} \\ \vdots \\ \{\epsilon, qn\} \\ \vdots \end{cases}$$

More generally, we can build an arena on any set X, whose moves are q (O move) and x (P move, for each $x \in X$). `nat` is just the instance of this construction where X is the set ω of natural numbers. Here are other instances:

- $X = \{T, F\}$. We use this arena to interpret the type `bool`.
- X is a singleton. We shall use this arena to interpret the type `comm` of commands in section 6 (renaming the O move as `run` and the unique P move as `done`).

- $X = \emptyset$. We shall call this arena $\mathtt{q}$: it has only one move: q. It is nevertheless useful, as we shall see.

We next consider the type $\mathtt{nat} \times \mathtt{nat}$. The corresponding arena is made of two disjoints copies of $\mathtt{nat}$, which we can write as $\mathtt{nat}_1$ and $\mathtt{nat}_2$, respectively. Thus, we have the following moves: q_1 and q_2 (O moves) and all moves n_1 and n_2 (P moves). By determinism, it is easy to see that a (meager) strategy can contain at most two non-empty plays (there are only two O moves, both initial), and the complete list of strategies is as follows (we also indicate the correspondence with the traditional denotational semantics of $\mathtt{nat} \times \mathtt{nat}$):

$$\left\{\begin{array}{lll} \{\epsilon\} & & \\ \ldots & \{\epsilon, q_1 m_1\} & \ldots \\ \ldots & \{\epsilon, q_2 n_2\} & \ldots \\ \ldots & \{\epsilon, q_1 m_1, q_2 n_2\} & \ldots \end{array}\right. \qquad \left\{\begin{array}{lll} (\bot, \bot) & & \\ \ldots & (m, \bot) & \ldots \\ \ldots & (\bot, n) & \ldots \\ \ldots & (m, n) & \ldots \end{array}\right.$$

But is this list complete? What about plays of the form $q_1 n_2$? Such plays are not allowed. To see why, we need to give a little more structure to the plays. We already mentioned that an arena specifies some causality relations between moves. For $\mathtt{nat}$, q is initial (not caused by any other move), and n is caused, or *enabled*, by q. We write $q \vdash n$. We impose always that when $m \vdash n$ then m and n have opposite *polarities* (i.e., m is an O move if and only if n is a P move), and also that initial moves are O moves. A play must be such that each move is *justified* by a previous move in the sequence, which enables it. At some point, it will be also important to record which precise occurrence in the sequence justifies each move, and we will thus consider sequences with pointers, or pointing sequences. For the moment, let us keep with ordinary sequences satisfying the above causality requirements, without recording the witnesses of causality as pointers. When forming $\mathtt{nat} \times \mathtt{nat}$, we do not add causalities other than those in each copy of $\mathtt{nat}$, i.e., q_1 and q_2 are initial, and every n_1 (resp. n_2) is enabled by q_1 (resp. q_2). Now we can see that $q_1 n_2$ is not a legal play, since n_2 is not justified (it can only be justified by q_2, which does not occur in the sequence). These explanations complete the discussion of this data type: the above list of strategies is indeed complete.

Our next data type is $\mathtt{nat} \to \mathtt{nat}$. Again, we take two copies $\mathtt{nat}_1$ and $\mathtt{nat}_2$ of $\mathtt{nat}$. But now we inverse the polarity of the moves of $\mathtt{nat}_1$, i.e., the O moves are q_2 and all n_1's, and the P moves are q_1 and all the moves n_2. The enablings are those inherited from each copy of $\mathtt{nat}$, i.e., q_1 enables n_1 and q_2 enables n_2, plus the following new enabling: $q_2 \vdash q_1$. Note that the change of polarity constraint is respected by the enabling relation, since q_1, as a move of $\mathtt{nat} \to \mathtt{nat}$, is a P move (also, for $q_1 \vdash n_1$, notice that *both* q_1 and n_1 have changed polarities). Let us examine which plays there are in this arena. $q_2 n_2$ is a legal play, and expresses a constant function of value n ($\lambda x.n$). How to express a non-constant (partial) function, then? Say, we want to interpret the function f which on input 0 returns

3 and on input 1 returns 4 and is otherwise undefined. We use the following syntax for such functions:

$$\lambda x.\texttt{case}\ x\ [0 \to 3, 1 \to 4]$$

Here is the strategy for this program:

$$\{\epsilon, q_2q_1, q_2q_1 0_1 3_2, q_2q_1 1_1 4_2\}$$

The play q_2q_1 is important: it expresses the fact that f is a function which looks at its argument before outputting anything. It is easy to check that these are the only two kinds of strategies in $\texttt{nat} \to \texttt{nat}$.

Let us stress the difference between the two sorts of strategies. There is another way to program a constant function, namely:

$$g = \lambda x.\texttt{case}\ x\ [0 \to n, \ldots, m \to n, \ldots]$$

with the corresponding strategy:

$$\{\epsilon, q_2q_1, q_2q_1 0_1 n_2, \ldots, q_2q_1 m_1 n_2, \ldots\}$$

or, in tree form:

$$q_2q_1 \begin{cases} 0_1 n_2 \\ \vdots \\ m_1 n_2 \\ \vdots \end{cases}$$

(it is standard to represent a tree as its set of branches).

Is g really constant? Not quite, since the function applied to $\perp$ yields $\perp$, not n. This is a *call-by-value* constant function, which retuns n only on an input which is a value, i.e., whose evaluation has been successful. But this raises the question of how to *apply* a strategy of type $\texttt{nat} \to \texttt{nat}$ to a strategy of type $\texttt{nat}$. The paradigm is that of computation-as-interaction. Suppose we apply f to $x = 0$ $((\lambda x.\texttt{case}\ x\ [0 \to 3, 1 \to 4])0)$. To avoid renamings of names, we consider 0 as a strategy of $\texttt{nat}_1$, namely as the tree

$$q_1 0_1$$

consisting of only one branch. We start playing in f, placing a token on q_2. Following strategy f, we then move to q_1. Now x enters the play. We place a second token on q_1 in x, so from now on there is a token in each strategy. Following x we move the x-token to 0_1. This shows the way to f, thus the f-token chooses the branch starting with 0_1 and then follows the strategy f and moves to 3_2, which is the final result, as expected, since, syntactically, we have:

$$(\lambda x.\texttt{case}\ x\ [0 \to 3, 1 \to 4])0 \to^* 3$$

using the β-rule $((\lambda x.M)N \to M[x \leftarrow N]$ and the rule

$$\texttt{case}\ i\ [\ldots, i \to M, \ldots] \to M\ .$$

Let us now compute $f2$, i.e., $x = 2$. The first steps of the interaction are the same, until the x-token reaches 2_1. Then the evaluation is stuck, because there is no corresponding 2_1 move to which the f-token could move. This is also conform to the syntactic computation, which gets stuck on $\texttt{case}\ 2\ [0 \to 3, 1 \to 4]$. In Scott semantics, this expression is interpreted as $\perp$. As a matter of fact, we can view $\texttt{case}\ x\ [0 \to 3, 1 \to 4]$ as a finite notation for

$$\texttt{case}\ x\ [0 \to 3, 1 \to 4, 2 \to \Omega, 3 \to \Omega, \ldots]$$

(where Ω is a syntactic counterpart of $\perp$), and then the syntactic evaluation of $f2$ yields Ω. As a final example, we contrast g and $\lambda x.n$ by applying both to Ω. The interaction for $(\lambda x.n)\Omega$ consists of moving the $(\lambda x.n)$-token from q_2 to n_2, thus no real interaction takes place, the function does not look at its argument! The interaction for $g\Omega$ ($x = \Omega$) is as follows: the g-token moves to q_1, thus we have to place a token on q_1 (in x), but we can't, since $x = \{\epsilon\}$. So the result is $\perp$ (the empty strategy).

Before we close the discussion on this type, let us come back to the close relation between syntax and (game) semantics. Let us consider again the program $\lambda x.\texttt{case}\ x\ [0 \to 3, 1 \to 4]$ and the strategy $\{\epsilon, q_2q_1,\ q_2q_1(0,1)(3,2), q_2q_1(1,1)(4,2)\}$, and let us represent them both in tree form (cf. above):

$$q_2q_1 \left\{ \begin{array}{l} 0_1 3_2 \\ 1_1 4_2 \end{array} \right. \qquad \lambda x.\ \texttt{case}\ x \left\{ \begin{array}{l} 0 \to 3 \\ 1 \to 4 \end{array} \right.$$

These are isomorphic representations! As a consequence, through the isomorphism, we can read syntax in terms of games. λx is an O move, $\texttt{case}\ x$ (the head variable) is a P move, 0 and 1 are O moves, 3 and 4 are P moves. Let us think of O moves as questions, and of P moves as answers. Then the program can be analyzed as follows:

- λx. Opponent (or observer) asks the program: "how do you behave?".
- $\texttt{case}\ x$. Player (or program) answers: "give me an input".
- 0. Opponent: "OK, I give you 0, how do you behave next?
- 3. Player: "output 3".

But the reader may ask: if programs and strategies are the same, why bother with games and strategies, why absorb a new (exotic) terminology? We can advocate two general kinds of reasons for making the effort:

1. Mathematics is full of such equivalent presentations: think of geometry versus analytic geometry, of matrices versus linear maps, etc... Some things are easier when one switches to the appropriate presentation.

2. The presentations may turn out to be equivalent in one setting, but not in a larger picture. It is precisely what happens with game semantics, once the constraints on strategies are relaxed.

Stuttering. By the isomorphism sketched above, the arena `nat` $\rightarrow$ `nat` contains "stuttering" strategies of the following kind:

$$q_\epsilon q_1 \left\{ \begin{array}{l} \vdots \\ 4_1 q_1 \left\{ \begin{array}{l} \vdots \\ 4_1 \\ \vdots \end{array} \right. \\ \vdots \end{array} \right. \quad \text{or even} \quad q_\epsilon q_1 \left\{ \begin{array}{l} \vdots \\ 4_1 q_1 \left\{ \begin{array}{l} \vdots \\ 3_1 2_\epsilon \\ \vdots \end{array} \right. \\ \vdots \end{array} \right.$$

corresponding to terms of the shape $\lambda x.\mathtt{case}\ x\ [\ldots, 4 \rightarrow \mathtt{case}\ x\ [\ldots], \ldots]$. Depending on which language we have in mind, it may or may not be desirable to identify these programs with respective simpler, non-stuttering ones (see Exercises 2 and 6). Such stuttering strategies do not exist in the model of sequential algorithms (see section 8 for more discussion on the difference between HO games and sequential algorithms). We just note here that the second stuttering strategy contains the play $q_\epsilon q_1 4_1 q_1 3_1 2_\epsilon$ whose projection on the input component $\mathtt{nat}_1$ is $q4q3$. Doesn't this contradict our discussion of type `nat`, in which we rejected such plays for non respecting innocence? No, it does not, because $q_1 4_1 q_1 3$ should be read as a subsequence of a play in $\mathtt{nat}_1 \rightarrow \mathtt{nat}_2$, in which O and P have been interchanged as regards the moves tagged with $_1$ (see Exercise 4.2.)

Now we consider the type $\mathtt{nat} \times \mathtt{nat} \rightarrow \mathtt{nat}$, which we tag as follows: $\mathtt{nat}_1 \times \mathtt{nat}_2 \rightarrow \mathtt{nat}_\epsilon$. We shall often use the convention that a type $\sigma_1 \rightarrow (\sigma_2 \rightarrow \ldots (\sigma_n \rightarrow \sigma) \ldots)$ or $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n \rightarrow \sigma$ (both types are "the same" up to currying and uncurrying) can be viewed as a tree

$$\sigma \left\{ \begin{array}{l} \sigma_1 \\ \vdots \\ \sigma_n \end{array} \right.$$

where in turn each $\sigma_i = \tau_{i1} \rightarrow \ldots \rightarrow \tau_{ij} \rightarrow \tau_i$ is a tree, etc... Note that we already applied this convention to the type $\mathtt{nat} \rightarrow \mathtt{nat}$. Once a type is converted to a tree like this, we tag each node of the tree with its address in the tree. Thus, in $\mathtt{nat}_1 \times \mathtt{nat}_2 \rightarrow \mathtt{nat}_\epsilon$, ϵ stand for the root, 1 stands for "left son", and 2 stands for "right son". The non-empty strategies of `nat` $\times$ `nat` $\to$ `nat` can start in three different ways: with $q_\epsilon n_\epsilon$, with $q_\epsilon q_1$, or with $q_\epsilon q_2$. There is only one family of strategies of the first kind: the constant functions. A strategy of, say, the second kind may look as follows:

$$q_\epsilon q_1 \left\{\begin{array}{l} \vdots \\ i_1 m_\epsilon \\ \vdots \\ j_1 q_2 \left\{\begin{array}{l} \vdots \\ k_2 n_\epsilon \\ \vdots \end{array}\right. \\ \vdots \end{array}\right.$$

and is the interpretation of the following program:

$$\lambda xy.\texttt{case}\ x\ [\ldots, i \to m, \ldots, j \to \texttt{case}\ y\ [\ldots, k \to n, \ldots], \ldots]$$

It is interesting to replace `nat` by `bool`, and to look at the various (non stuttering) strategies for computing the disjunction. There are four of them, corresponding to four programs which we encourage the reader to write down:

$$lor = q_\epsilon q_1 \left\{\begin{array}{l} F_1 q_2 \left\{\begin{array}{l} F_2 F_\epsilon \\ T_2 T_\epsilon \end{array}\right. \\ T_1 T_\epsilon \end{array}\right. \qquad ror = q_\epsilon q_2 \left\{\begin{array}{l} F_2 q_2 \left\{\begin{array}{l} F_1 F_\epsilon \\ T_1 T_\epsilon \end{array}\right. \\ T_2 T_\epsilon \end{array}\right.$$

$$lror = q_\epsilon q_1 \left\{\begin{array}{l} F_1 q_2 \left\{\begin{array}{l} F_2 F_\epsilon \\ T_2 T_\epsilon \end{array}\right. \\ T_1 q_2 \left\{\begin{array}{l} F_2 T_\epsilon \\ T_2 T_\epsilon \end{array}\right. \end{array}\right. \qquad rlor = q_\epsilon q_2 \left\{\begin{array}{l} F_2 q_2 \left\{\begin{array}{l} F_1 F_\epsilon \\ T_1 T_\epsilon \end{array}\right. \\ T_2 q_1 \left\{\begin{array}{l} F_1 T_\epsilon \\ T_1 T_\epsilon \end{array}\right. \end{array}\right.$$

The Scott semantics of these four programs/strategies consist of three functions only, because the two strategies *lror* and *rlor* have the same behaviour when applied (or interacting with) a strategy (x, y) of `nat`$\times$`nat`: the result is $\perp$ unless both x and y are different from $\perp$, and otherwise the result is as in the classical disjunction. The other programs can be separated, by applying them to an input on which they yield different results:

$$\begin{array}{ll} lor(1,\perp) = 1 & ror(1,\perp) = rlor(1,\perp) = \perp \\ ror(\perp,1) = 1 & lror(\perp,1) = \perp \end{array}$$

We shall see soon how to separate *lror* and *rlor*. Let us also notice that Scott semantics allows for yet another disjunction, the function *por* such that $por(\perp,1) = por(1,\perp) = 1$.

There is no sequential program, i.e., no λ-term (extended with `case` statements as above) whose interpretation in Scott semantics is the function *por*. Summarizing, there are two interesting features of HO (and AJM) games:

- Every strategy is the interpretation of a program.This interpretation is injective for λ-terms with `case` statements in suitable canonical form, and we shall prove that it is even bijective, with the right restrictions on strategies, namely innocence, already mentioned, and another condition called well-bracketing.
- Games allow us to distinguish programs according to the way they compute, e.g. *lror* and *rlor* differ only by the order in which they look at the two arguments.

It is also interesting to replace `nat` with `q`. There are only three strategies in the arena $\mathtt{q} \times \mathtt{q} \to \mathtt{q}$ (well, it is actually amazing that there are some, this is really here that something very different from Scott semantics is happening), namely:

$$\{\epsilon\} \qquad \{\epsilon, q_\epsilon q_1\} \qquad \{\epsilon, q_\epsilon q_2\}$$

Hence (up to renaming) $\mathtt{q} \times \mathtt{q} \to \mathtt{q}$ is `bool`!

Let us now consider the type $\mathtt{nat} \to (\mathtt{nat} \times \mathtt{nat})$. Let us tag it as $\mathtt{nat}_1 \to (\mathtt{nat}_2 \times \mathtt{nat}_3)$. A (meager, innocent) strategy of this type consists of the union (as sets of plays) of one strategy f of $\mathtt{nat}_1 \to \mathtt{nat}_2$ and one strategy g of $\mathtt{nat}_1 \to \mathtt{nat}_3$. Hence it is a forest of two trees. Let us give one example: $\lambda x.(x+1, x+2)$ is interpreted as:

$$\left\{ \begin{array}{l} q_2 q_1 \left\{ \begin{array}{l} \vdots \\ n_1 (n+1)_2 \\ \vdots \end{array} \right. \\ q_3 q_1 \left\{ \begin{array}{l} \vdots \\ n_1 (n+2)_3 \\ \vdots \end{array} \right. \end{array} \right.$$

We now consider higher-order types. Let us take the simplest one: $(\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$, tagged as $(\mathtt{nat}_{11} \to \mathtt{nat}_1) \to \mathtt{nat}_\epsilon$. Here, the intuitions are not so easy to follow, hence we prefer to rely on syntax and follow the bijection sketched above. Consider the following program of this type:

$$h = \lambda f.\mathtt{case}\ f3\ [4 \to 7, 6 \to 9]$$

We adopt the following tree representation for this program:

$$\lambda f.\ \mathtt{case}\ f \left\{ \begin{array}{l} (3) \\ 4 \to 7 \\ 6 \to 9 \end{array} \right.$$

In this representation, we have made the decision to consider that (3), $4 \to 7$ and $6 \to 9$ are at the same level: (3) is the first (and unique) argument of f, and $4 \to 7$ and $6 \to 9$ are continuations. Let us see why this representation is appropriate. We first translate h to a strategy:

$$q_\epsilon q_1 \left\{ \begin{array}{l} q_{11} 3_{11} \\ 4_1 7_\epsilon \\ 6_1 9_\epsilon \end{array} \right.$$

We read it as follows. The play $q_\epsilon q_1$ gives us the information that h wants to know its input f. This input can be:

- A constant function $\lambda x.n$. The play $q_\epsilon q_1 4_1 7_\epsilon$ (resp. the play $q_\epsilon q_1 6_1 9_\epsilon$) indicates that if $n = 4$ (resp. $n = 6$), then h outputs 7 (resp. 9). This agrees with syntactic evaluation, for example:

$$h(\lambda x.4) \to^* 7$$

- A strategy which itself wants to know its input. The play $q_\epsilon q_1 q_{11} 3_{11}$ indicates that the relevant information for h is $f3$.

The above description of h as a *meager* strategy is complete. Let us describe a few evaluations hf, in terms of interactions. First, consider again $f = \lambda x.4$, i.e., $f = q_1 4_1$: the h-token moves from q_ϵ to q_1, then the f-token is placed on q_1 and moves to 4_1, so the f-token moves to 4_1, and then to 7_ϵ. If $f = \lambda x.5$, then the interaction gets stuck after the x-token has moved to 5_1, because there is no corresponding move 5_1 to which the h-token could move. It is not difficult to see that the evaluation will also get stuck if $f = \lambda x.\mathtt{case}\ x\ [0 \to p]$. This is because this input f does not give any information on $f3$. The evaluation will also get stuck (after a longer interaction) if $f = \mathtt{case}\ x\ [3 \to 5]$, because when $f3$ is not 4 or 6, h is not prepared to continue. Finally, let us take, say, $f = \lambda x.\mathtt{case}\ x\ [0 \to 3, 3 \to 6]$, or in tree and strategy form:

$$q_1 q_{11} \left\{ \begin{array}{l} 0_{11} 3_1 \\ 3_{11} 6_1 \end{array} \right.$$

Then the syntactic evaluation is successful:

$$\begin{array}{rcl} hf & \to & \mathtt{case}\ (\lambda x.\mathtt{case}\ x\ [0 \to 3, 3 \to 6])3\ [4 \to 7, 6 \to 9] \\ & \to & \mathtt{case}\ (\mathtt{case}\ 3\ [0 \to 3, 3 \to 6])\ [4 \to 7, 6 \to 9] \\ & \to & \mathtt{case}\ 6\ [4 \to 7, 6 \to 9] \\ & \to & 9 \end{array}$$

Let us now describe the steps of the interaction:

h	f	interaction
$\underline{q_\epsilon}$		q_ϵ
$q_\epsilon q_1$	$\underline{q_1}$	$q_\epsilon q_1$
$\underline{q_\epsilon q_1 q_{11}}$	$q_1 q_{11}$	$q_\epsilon q_1 q_{11}$
$q_\epsilon q_1 q_{11} 3_{11}$	$\underline{q_1 q_{11} 3_{11}}$	$q_\epsilon q_1 q_{11} 3_{11}$
$\underline{q_\epsilon q_1 q_{11} 3_{11} 6_1}$	$q_1 q_{11} 3_{11} 6_1$	$q_\epsilon q_1 q_{11} 3_{11} 6_1$

In the first two columns, we have displayed the progression of the tokens in the two strategies. Alternatively, we may consider (third column) a unique *interaction sequence*, where each move tagged with $_1$ or $_{11}$ is considered both as a move of $A \to \mathtt{nat}_\epsilon$ and of A (where $A = \mathtt{nat}_{11} \to \mathtt{nat}_1$). The underlined plays are the *active* plays, i.e., the odd plays which are extended according to the respective strategies. Let us examine the last line. We have reached the play $q_\epsilon q_1 q_{11} 3_{11} 6_1$, which does not enter into our description of h. However, the view (cf. above) of this play is simply $q_\epsilon q_1 6_1$ (because 6_1 is justified by q_1), and, by innocence, we know that (the fat representation of) h contains also $q_\epsilon q_1 q_{11} 3_{11} 6_1 9_\epsilon$ since it contains $q_\epsilon q_1 6_1 9_\epsilon$. Thus the interaction terminates with the expected result 9, as follows:

$\underline{q_\epsilon q_1 q_{11} 3_{11} 6_1}$	$q_1 q_{11} 3_{11} 6_1$	$q_\epsilon q_1 q_{11} 3_{11} 6_1$
$q_\epsilon q_1 q_{11} 3_{11} 6_1 9_\epsilon$	$q_1 q_{11} 3_{11} 6_1$	$q_\epsilon q_1 q_{11} 3_{11} 6_1 9_\epsilon$

The evaluation mechanism that we have sketched on this example follows the rules of an abstract machine, called the View Abstract Machine, whose description can be found in, say, [32] (it goes back to [24]).

We now consider the type $(\mathtt{nat}_{11} \times \mathtt{nat}_{12} \to \mathtt{nat}_1) \to \mathtt{nat}_\epsilon$. The following is a strategy which is not expressible in Scott semantics, and which is not the interpretation/counterpart of a λ-term with `case` statements:

$$q_\epsilon q_1 \begin{cases} q_{11} 0_\epsilon \\ q_{12} 1_\epsilon \\ \vdots \\ n_1 (n+2)_\epsilon \\ \vdots \end{cases}$$

We call this strategy `catch`. It has the following behaviour: `catch` looks at its input f, and examines whether f is a constant function, or a function whose first action is to look at its first (resp. second) argument. This is a *control* operator, which can be used to encode some exception mechanisms (see [66]). It can be expressed in the syntax via the following rules:

$$\begin{array}{l}\texttt{catch}(\lambda xy.\texttt{case}\ x\ [\ldots]) \to 0\\ \texttt{catch}(\lambda xy.\texttt{case}\ y\ [\ldots]) \to 1\\ \texttt{catch}(\lambda xy.n) \to n+2\end{array}$$

For example, we have:

$$\texttt{catch}(\mathit{lror}) \to 0 \qquad \texttt{catch}(\mathit{rlor}) \to 1$$

Thus `catch` increases our separating power (remember that Scott semantics cannot distinguish *lror* and *rlor*).

Pointers. Consider the two terms

$$\begin{array}{l}\mathit{Kierstead}_1 = \lambda f.\texttt{case}\ f(\lambda x.\texttt{case}\ f(\lambda y.\texttt{case}\ x))\\ \mathit{Kierstead}_2 = \lambda f.\texttt{case}\ f(\lambda x.\texttt{case}\ f(\lambda y.\texttt{case}\ y))\end{array}$$

(named after Kierstead, quoted by Kleene) of type $((\texttt{bool}_{111} \to \texttt{bool}_{11}) \to \texttt{bool}_1) \to \texttt{bool}_\epsilon$, in which $\texttt{case}\ M$ stands everywhere for $\texttt{case}\ M\ [T \to T, F \to F]$. The two associated "strategies" are the same:

$$q_\epsilon q_1 \left\{\begin{array}{l} q_{11}q_1 \left\{\begin{array}{l} q_{11}q_{111} \left\{\begin{array}{l} T_{111}T_{11}\\ F_{111}F_{11}\end{array}\right.\\ T_1T_{11}\\ F_1F_{11}\end{array}\right.\\ T_1T_\epsilon\\ F_1F_\epsilon\end{array}\right.$$

The difficulty lies in the view $q_\epsilon q_1 q_{11} q_1 q_{11} q_{111}$. The last move q_{111} is justified by q_{11}, but there are two occurrences of q_{11} in the view, and the (only) difference between the (intepretations of the) two terms lies there.

This leads to the following two strategies, where we make explicit all the pointers from P moves to O moves. We do not need to write explicitly the pointers from O moves to P moves since we are only listing the views of the strategies associated to the respective terms (but see Exercise 2.1). We use the notation $[m, \overset{i}{\hookleftarrow}]$ for a P move, where i is a natural number recording how many O moves one must pass until meeting the justifier. The only difference between the two strategies lies in the underlined moves $[q_{111}, \overset{1}{\hookleftarrow}]$ and $[q_{111}, \overset{0}{\hookleftarrow}]$, respectively.

$$q_\epsilon[q_1,\overset{0}{\hookleftarrow}]\left\{\begin{array}{l} q_{11}[q_1,\overset{1}{\hookleftarrow}]\left\{\begin{array}{l} q_{11}\underline{[q_{111},\overset{1}{\hookleftarrow}]}\left\{\begin{array}{l} T_{111}[T_{11},\overset{1}{\hookleftarrow}] \\ F_{111}[F_{11},\overset{1}{\hookleftarrow}] \end{array}\right. \\ T_1[T_{11},\overset{1}{\hookleftarrow}] \\ F_1[F_{11},\overset{1}{\hookleftarrow}] \end{array}\right. \\ T_1[T_\epsilon,\overset{1}{\hookleftarrow}] \\ F_1[F_\epsilon,\overset{1}{\hookleftarrow}] \end{array}\right.$$

$$q_\epsilon[q_1,\overset{0}{\hookleftarrow}]\left\{\begin{array}{l} q_{11}[q_1,\overset{1}{\hookleftarrow}]\left\{\begin{array}{l} q_{11}\underline{[q_{111},\overset{0}{\hookleftarrow}]}\left\{\begin{array}{l} T_{111}[T_{11},\overset{1}{\hookleftarrow}] \\ F_{111}[F_{11},\overset{1}{\hookleftarrow}] \end{array}\right. \\ T_1[T_{11},\overset{1}{\hookleftarrow}] \\ F_1[F_{11},\overset{1}{\hookleftarrow}] \end{array}\right. \\ T_1[T_\epsilon,\overset{1}{\hookleftarrow}] \\ F_1[F_\epsilon,\overset{1}{\hookleftarrow}] \end{array}\right.$$

The pointers from question moves q_1 and q_{111} are just the ones given by the corresponding binding λ's, and our encoding of pointers is then nothing else than De Bruijn indices (see e.g. [1]). The pointers from answer moves $T_\epsilon, F_\epsilon, T_{11}, F_{11}$ are all to the last question which remains open. This is the well-bracketing condition to which we already alluded, and which we shall further discuss in section 5.

Geometric Abstract Machine. Let us come back to the token machine kind of execution described earlier in this section. Let us describe the interaction of

$$h = q_\epsilon[q_1,\overset{0}{\hookleftarrow}]\left\{\begin{array}{l} q_{11}[3_{11},\overset{0}{\hookleftarrow}] \\ 4_1[7_\epsilon,\overset{1}{\hookleftarrow}] \\ 6_1[9_\epsilon,\overset{1}{\hookleftarrow}] \end{array}\right. \quad \text{and} \quad f = q_1[q_{11},\overset{0}{\hookleftarrow}]\left\{\begin{array}{l} 0_{11}[3_1,\overset{1}{\hookleftarrow}] \\ 3_{11}[6_1,\overset{1}{\hookleftarrow}] \end{array}\right.$$

(where we have made the pointers explicit) in this style. We mark the opponent moves with numbers corresponding to the progression of the exploration of the machine:

$$\langle q_\epsilon,\mathbf{1}\rangle[q_1,\overset{0}{\hookleftarrow}]\left\{\begin{array}{l} \langle q_{11},\mathbf{3}\rangle[3_{11},\overset{0}{\hookleftarrow}] \\ \langle 6_1,\mathbf{5}\rangle[9_\epsilon,\overset{1}{\hookleftarrow}] \end{array}\right.$$

$$\langle q_1,\mathbf{2}\rangle[q_{11},\overset{0}{\hookleftarrow}]\left\{\begin{array}{l} \langle 3_{11},\mathbf{4}\rangle[6_1,\overset{1}{\hookleftarrow}] \end{array}\right.$$

Up to the fourth step of the machine, things happen as described before: one progresses in one branch of each of the two strategies. But step 5 breaks this. This is because the *meager* representation of h does not contain the play $q_\epsilon[q_1,\overset{0}{\hookleftarrow}]q_{11}[3_{11},\overset{0}{\hookleftarrow}]6_1[9_1,\overset{2}{\hookleftarrow}]$, but only its view $q_\epsilon[q_1,\overset{0}{\hookleftarrow}]6_1[9_1,\overset{1}{\hookleftarrow}]$. The View Abstract Machine computes such “fat plays” of f on demand. But there is a way to stick with views only, and to extend the token machinery accordingly. After a move $\mathbf{n}$ in one of the strategies, the (unique) P move that follows it has the form $[a,\overset{i}{\hookleftarrow}]$. As we have seen, the information a serves us to choose a branch in the other strategy, while the second information will tell us where this choice will happen. To describe this mechanism concisely, it is useful to denote with $(n+1)'$ the moment where the P move below the move (played at time) $\mathbf{n}$ is visited. Then the rule is as follows:

- If n' points to $\mathbf{m}$, then $\mathbf{n}$ should be played under m' (and this move should be a if n' was $[a,\overset{i}{\hookleftarrow}]$).

In our example, we have that $5'$ points to $\mathbf{2}$, hence we play $\mathbf{5}$ under $2'$.

As a more sophisticated example, we show the interaction of $\mathit{Kierstead}_1$ with

$$\lambda g.\texttt{case}\ g(\texttt{case}\ gT\ [T\to T,F\to F])\ [T\to F,F\to T]$$

or, in strategy form:

$$q_1[q_{11},\overset{0}{\hookleftarrow}]\begin{cases} q_{111}[q_{11},\overset{1}{\hookleftarrow}]\begin{cases} q_{111}[F_{111},\overset{0}{\hookleftarrow}]\\ T_{11}[T_{111},\overset{1}{\hookleftarrow}]\\ F_{11}[F_{111},\overset{1}{\hookleftarrow}]\end{cases}\\ T_{11}[F_1,\overset{1}{\hookleftarrow}]\\ F_{11}[T_1,\overset{1}{\hookleftarrow}]\end{cases}$$

Here is the trace of the execution:

$$\langle q_\epsilon,\mathbf{1}\rangle[q_1,\overset{0}{\hookleftarrow}]\begin{cases} \langle q_{11},\mathbf{3}\rangle[q_1,\overset{1}{\hookleftarrow}]\begin{cases}\langle q_{11},\mathbf{5}\rangle[q_{111},\overset{1}{\hookleftarrow}]\left\{\ \langle T_{111},\mathbf{15}\rangle[T_{11},\overset{1}{\hookleftarrow}]\right.\\ \langle F_1,\mathbf{17}\rangle[F_{11},\overset{1}{\hookleftarrow}]\end{cases}\\ \langle q_{11},\mathbf{7}\rangle[q_1,\overset{1}{\hookleftarrow}]\begin{cases}\langle q_{11},\mathbf{9}\rangle[q_{111},\overset{1}{\hookleftarrow}]\left\{\ \langle F_{111},\mathbf{11}\rangle[F_{11},\overset{1}{\hookleftarrow}]\right.\\ \langle T_1,\mathbf{13}\rangle[T_{11},\overset{1}{\hookleftarrow}]\end{cases}\\ \langle T_1,\mathbf{19}\rangle[T_\epsilon,\overset{1}{\hookleftarrow}]\end{cases}$$

$$\left\{\begin{array}{l} \langle q_1, \mathbf{2}\rangle[q_{11}, \overset{0}{\hookleftarrow}] \left\{\begin{array}{l} \langle q_{111}, \mathbf{6}\rangle[q_{11}, \overset{1}{\hookleftarrow}] \left\{\begin{array}{l} \langle q_{111}, \mathbf{10}\rangle[F_{111}, \overset{0}{\hookleftarrow}] \\ \langle T_{11}, \mathbf{14}\rangle[T_{111}, \overset{1}{\hookleftarrow}] \end{array}\right. \\ \langle F_{11}, \mathbf{18}\rangle[T_1, \overset{1}{\hookleftarrow}] \end{array}\right. \\ \langle q_1, \mathbf{4}\rangle[q_{11}, \overset{0}{\hookleftarrow}] \left\{ \langle T_{11}, \mathbf{16}\rangle[F_1, \overset{1}{\hookleftarrow}] \right. \\ \langle q_1, \mathbf{8}\rangle[q_{11}, \overset{0}{\hookleftarrow}] \left\{ \langle F_{11}, \mathbf{12}\rangle[T_1, \overset{1}{\hookleftarrow}] \right. \end{array}\right.$$

In this example, we face a new (and the last) difficulty in the description of the token machine. A node may be visited several times, and one needs to keep track of this, by opening new copies of subtrees as the need arises. We do not describe this formally, but the reader can see what this amounts to in the above execution. The resulting machine is called the Geometric Abstract Machine. As far as we know, it is described for the first time in [30] (see [32] for a proof of equivalence with the View Abstract Machine execution).

Strong evaluation. Our computing devices (View Abstract Machine or Geometric Abstract Machine) are not sufficient for computing functional values. Consider the following term k:

$$(\lambda xy.\texttt{case}\ y\ [6 \to \texttt{case}\ x\ [4 \to 8, 2 \to 1], 3 \to \texttt{case}\ x\ [4 \to 5, 1 \to 6]])4$$

This term reduces to

$$\lambda y.\texttt{case}\ y\ [6 \to 8, 3 \to 5]$$

The interaction is thus between the following two strategies:

$$q_\epsilon q_2 \left\{\begin{array}{l} 6_2 q_1 \left\{\begin{array}{l} 4_1 8_\epsilon \\ 2_1 1_\epsilon \end{array}\right. \\ 3_2 q_1 \left\{\begin{array}{l} 4_1 5\epsilon \\ 1_1 6_\epsilon \end{array}\right. \end{array}\right.$$

$$q_1 4_1$$

When we start our machinery of moving tokens, we stop after we have read the play $q_\epsilon q_2$, because there is nobody to tell us which branch to choose on the function strategy. In order to get further plays of k we need relaunch evaluation. The hand is given back to the user after q_2: this corresponds to the head information about $\lambda y.\texttt{case}\ y\ [6 \to 8, 3 \to 5]$, namely "$\lambda y.\texttt{case}\ y\ [\ldots]$" (or "the result is a function that wants to look at its (unique) argument"). If you want to know more, you have to supply a specific question, say "what if $y = 6$?". This prompts a new evaluation session: one extends $q_\epsilon q_2$ to $q_\epsilon q_2 6_2$, then the interaction goes on to $q_\epsilon q_2 6_2 q_1$ and then (passing the hand to the argument 4) to $q_\epsilon q_2 6_2 q_1 4_1$ and finally to $q_\epsilon q_2 6_2 q_1 4_1 8_\epsilon$. Hiding the intermediate type $\mathtt{nat}_1$, we have obtained the following view of the result: $q_\epsilon q_2 6_2 8_\epsilon$. One computes similarly the other view $q_\epsilon q_2 3_2 5_\epsilon$ of the result, which is thus, as expected:

$$q_\epsilon q_2 \left\{ \begin{array}{l} 6_2 8_\epsilon \\ 3_2 5_\epsilon \end{array} \right.$$

The slogan here is that strong evaluation is a lazy, stream-like loop of evaluation.

A trade-off between tags and trees. In most papers on game semantics, tags like $_\epsilon$, $_1$, $_{11}$, etc... are avoided by presenting plays using two dimensions, as in the self-explanatory table below, which is a tag-free description of $q_\epsilon q_1 q_{11} 3_{11} 6_1 9_\epsilon$:

$\mathtt{nat}_{11}$	$\mathtt{nat}_1$	$\mathtt{nat}$
		q
	q	
q		
3		
	6	
		9

This notation is pleasant, but makes it hard to have a global view on the strategy. Another disadvantage is that it takes a lot of space... In these notes, we have chosen to favour the strategy trees, taking the tags as a “mal nécessaire”, which by the way is the “pain quotidien” of programmers.

Exercise 2.1 *Describe (again) the interaction between Kierstead$_1$ and*

$$\lambda g.\mathtt{case}\ g(\mathtt{case}\ gF\ [T \to T, F \to F])\ [T \to F, F \to T]$$

expressed now in the machinery of the view abstract machine. (This exercise is important, because it shows how the pointer information from P moves to O moves in the function becomes an information from O moves to P moves from the point of view of the argument and hence becomes relevant in the computation of views of the argument, whence the difference of execution when one takes Kierstead$_2$ instead of Kierstead$_1$.) Describe the interaction resulting from replacing Kierstead$_1$ by Kierstead$_2$, using either machine you prefer.

Exercise 2.2 *Show that Kierstead$_1$ and Kierstead$_2$ can (also, cf. Exercise 2.1) be separated by applying them to (a version of)* `catch`*. Describe the corresponding interactions.*

Exercise 2.3 *Exhibit two non stuttering strategies having the same input-output behaviour as each of the above stuttering programs, respectively.*

3 PCF and extensions

We step momentarily aside from moves, strategies, etc... and study a small core functional programming language, PCF, that was defined in the first works on denotational semantics (Scott, Plotkin [96, 94]). Most of the examples of section 2 were already written in this language

We recall the use of a fixpoint operator Y for writing recursive definitions, with the corresponding self-explanatory syntactic sugar:

$$\begin{array}{l}\texttt{let modulo}(x,y) = \texttt{if } x < y \texttt{ then } x \texttt{ else modulo}(x-y,y)\\ Y(\lambda f.\lambda x.\lambda y.\texttt{if } x<y \texttt{ then } x \texttt{ else } f(x-y)y)\end{array}$$

Here is the complete syntax of PCF:

$$M ::= x \mid \lambda x.M \mid MM \mid n \mid T \mid F \mid \texttt{pred} \mid \texttt{succ} \mid \texttt{zero?} \mid \texttt{case}\ M\ [\ldots, c \to M, \ldots] \mid Y$$

In the `case` construction, the c's are all distinct, and are either all natural numbers, or all boolean values. Another notation for $\texttt{case}\ M\ [T \to N, F \to P]$ is $\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P$.

Conventions. We write $M(NP) = MNP$, $\lambda x.(\lambda y.M) = \lambda xy.M$, $A \to (B \to C) = A \to B \to C$.
All terms must be well typed. The syntax of types is the following:

$$A ::= \texttt{bool} \mid \texttt{nat} \mid A \to A$$

The typing judgments have the form $\Gamma \vdash M : A$, where Γ is called the environment (or context) and consists in declarations $x_1 : A_1, \ldots, x_n : A_n$.

Operational semantics. The computation of terms may be specified in a variety of ways:

- Reduction, given by rewrite rules that may be applied to any appropriate subterm (or redex).
- "Small step": here, one specifies a determistic reduction strategy that says at each step how to go to the next step.
- "Big step": here, one specifies directly the final value of the program.

In these notes, we consider reduction. Here are some of the rules:

$$\begin{array}{l}
(\lambda x.M)N \longrightarrow M[x \leftarrow N] \\
YM \longrightarrow M(YM) \\
\texttt{succ}\, 0 \longrightarrow 1 \\
\texttt{pred}\, 0 \longrightarrow 0 \\
\texttt{pred}\, 1 \longrightarrow 0 \\
\texttt{zero?}\, 0 \longrightarrow T \\
\texttt{zero?}\, 2 \longrightarrow F \\
\texttt{case}\; c\, [\ldots, c \to M, \ldots] \longrightarrow M
\end{array}$$

Examples:

$$\begin{array}{lll}
(\lambda x.x+3)4 & \longrightarrow & 7 \\
 & & \\
YM32 & \longrightarrow & M(YM) \qquad (M = \lambda f.\lambda x.\lambda y.\texttt{if}\ x < y\ \texttt{then}\ x\ \texttt{else}\ f(x-y)y) \\
 & \longrightarrow^{\star} & \texttt{if}\ 3 < 2\ \texttt{then}\ 3\ \texttt{else}\ YM(3-2)2 \\
 & \longrightarrow^{\star} & YM12 \\
 & \longrightarrow^{\star} & \texttt{if}\ 1 < 2\ \texttt{then}\ 1\ \texttt{else}\ YM(3-2)2 \\
 & \longrightarrow^{\star} & 1
\end{array}$$

All the rules (or axioms) can be applied in any context C. Contexts are terms of the following syntax:

$$C ::= \ldots \mid [\,]$$

where the ... repeat all the constructions of PCF. Informally, a context is a term of PCF with a hole (see also Exercise 3.1). Given C, M (where M has the same type as the hole of C), $C[M]$ is the term obtained by replacing $[\,]$ by M (for example, if $C = \lambda x.[\,]$, then $C[x] = \lambda x.x$).

A normal form is a term M such that there does not exist N such that $M \longrightarrow N$. Here is a simple example of a computation:

$$\texttt{case}\ (\texttt{pred}\, 2)\ [1 \to T] \longrightarrow \texttt{case}\ 1\ [1 \to T] \longrightarrow T$$

Denotational semantics. Here one assigns meanings to programs in some mathematical structures. Some of the traditional key features of denotational semantics are the following:

- It does not want to speak directly about computation by stepwise rewriting to a normal form. In particular, it is invariant under reduction, i.e., if $M \Longrightarrow N$, then $[\![M]\!] = [\![N]\!]$. This property is called soundness.
- It is *compositional*: if $[\![M]\!] = [\![N]\!]$ then $[\![C[M]]\!] = [\![C[N]]\!]$.

The most simpe way of "thinking denotationally" is to read a statement $\Gamma \vdash M : A$ as "M is a function (more generally a morphism) from Γ to A". Base types are given an interpretation, and the $\rightarrow$ construction is interpreted by a (categorical) exponent construction: $[\![A_1 \rightarrow A_2]\!] = [\![A_2]\!]^{[\![A_1]\!]}$, the latter being also written $[\![A_1]\!] \rightarrow [\![A_2]\!]$. One often writes A for $[\![A]\!]$.

Computational Adequacy. A *program* is a term P that is *closed* and *of base type.* Any "reasonable" or *standard* model (that interprets `nat` as the natural numbers etc...) satisfies the following property, called computational adequacy: for any program P,

$$P \longrightarrow^{\star} c \quad \text{iff} \quad [\![P]\!] = c$$

The "only if" direction is an immediate consequence of the soundness of the model, while the "if" direction is proved using a logical relation technique [94].

Observational equivalence. We write $M =_{obs} N$ ("M is observationally, or operationally, equivalent to N") iff (by definition) for all C such that $C[M]$ and $C[N]$ are programs, we have

$$C[M] \longrightarrow^{\star} c \quad \text{iff} \quad C[N] \longrightarrow^{\star} c$$

Base types and base type constants T, F, n, are called observable types and values, respectively.

Adequacy. We want to compare the denotational and observational equalities. One direction, called adequacy, is a consequence of computational adequacy.

Proposition 3.1 $[\![M]\!] = [\![N]\!] \quad \Rightarrow \quad M =_{obs} N.$

PROOF. If $C[M] \longrightarrow^{\star} c$, then $[\![C[M]]\!] = c$. But we have $[\![C[M]]\!] = [\![C[N]]\!]$ (compositionality of the denotational semantics), hence $[\![C[N]]\!] = c$, from which $C[N] \longrightarrow^{\star} c$ follows (by the difficult side of computational adequacy). □

Full abstraction. A model is fully abstract when

$$[\![M]\!] = [\![N]\!] \quad \text{iff} \quad M =_{obs} N$$

The "if" direction is the difficult one, and corresponds to the *full* of full abstraction or the *complète* of "complète adéquation" (the French term used for full abstraction). A nice explanation of why this is the hard side of the equivalence comes from the theory of intersection types [23] and of "domains in logical form" [2], which recasts the denotational side of the correspondence as the side of "proofs", and the operational side as the side of "models". Then, under these glasses, fullness/completeness goes in the right direction: if $\models M = N$ (i.e., if $M =_{op} N$), then $\vdash M = N$ (i.e., $[\![M]\!] = [\![N]\!]$).

Given a language and a model that is only adequate, in order to obtain a full abstraction result, one may try two directions:

- Change (enrich) the language. This is probably the most interesting from the point of view of applications, as it may lead to "semantics guided" language design. Here are two examples:
  - (Plotkin 77 [94]) The continuous model (Scott, Plotkin) of PCF is not fully abstract. But adding the "parallel or" (see below), the same model becomes fully abstract (for PCF+`por`, thus).
  - (Cartwright, Curien, Felleisen 92 [20]) The model of sequential algorithms [13] of PCF is not fully abstract. Adding `catch` (cf. section 2), the same model becomes fully abstract (for PCF+`catch`, thus).
- Change the model. For example, what about a fully abstract model of PCF for short (no extension!)? Milner (1977 [89]) has constructed such a model, essentially as a quotient of a model of terms. Roughly, the quotient is performed in two passes: one takes the terms modulo the above equations (for example, $YM = M(YM)$, etc..), and then one performs a much more "violent" quotient by identifying terms M, N such that $M =_{obs} N$ (see examples below). Clearly, it is not terribly surprising that full abstraction is obtained in this way, because one has put on the "denotational" side what was on the operational one.

Definability, separability, extensionality. The search for a "direct" construction of a fully abstract model of PCF (that is, without making such a quotient), is at the origin of a wealth of works that have been interesting for themselves. Before we come to that, we examine general sufficient conditions that guarantee full abstraction. We assume for simplicity that there is only one base type o, but of course the same holds in presence of several base types (thus, for PCF). An element h of (the interpretation of) type A is called definable if there exists a closed term M of type A such that $[\![M]\!] = h$.

Proposition 3.2 *If the model is such that for every distinct f, g of the same type A (interpreting some syntactic type) there exists a* definable *h of type $A \to o$ such that $hf \neq hg$, then it is fully abstract.*

PROOF. If $[\![M]\!] \neq [\![N]\!]$, let h be given by our assumption, and let P be such that $[\![P]\!] = h$, $v_1 = h([\![M]\!])$, $v_2 = h([\![N]\!])$, and $C = P\,[\,]$. Then $C[M] \to^* v_1$ and $C[M] \to^* v_2$, and hence $M \neq_{op} N$. $\square$

We shall call this sufficient property for full abstraction the *definable separability* property. To be best of our knowledge, the first proofs of full abstraction based on definable separability were:

- (in an untyped setting) the one relating the model D_∞ of the untyped lambda-calculus and the notion of head normal form [59, 99] (see [31] for a concise and "interactive" account), and
- in the present typed setting, the proof of full abstraction of the model of sequential algorithms with respect to PCF+`catch`, already quoted above.

Proposition 3.3 *Under the assumptions that the type hierarchy is the simple type hierarchy and that the model is enriched over algebraic complete partial orders (for this notion, see e.g. [9]), a sufficient condition for definable separability (and hence for full abstraction) is the conjunction of the following two properties of the model:*

1. compact definability*: all compact elements at all types are definable, and*
2. extensionality*: the elements of the model at function types are functions (in category-theoretical terms, the model has enough points).*

PROOF. We may write an arbitrary type A as $A_1 \to \ldots \to A_n \to o$. Let $f \neq g$. By extensionality and algebraicity, there exist compact $d_1, \ldots, d_n$ such that $fd_1 \ldots d_n \neq gd_1 \ldots d_n$. Let P_i be such that $d_i = [\![P_i]\!]$ $(1 \leq i \leq n)$. Let $h = [\![\lambda x.xP_1 \ldots P_n]\!]$. Then $hf \neq hg$. $\square$

Of course, one can prove directly (i.e., without going through definable separability) that an extensional model enjoying the definability property is fully abstract (in the proof above, instead of defining h, one directly defines the context $[\,]\,P_1 \ldots P_n$). This is how Plotkin proved that the continuous model is fully abstract for PCF+`por`.

Let us make here two short digressions.

- In [94], a further extension of PCF is also presented, such that every *computable* element of the model at all types becomes definable – a property which Plotkin called *universality.*
- In works on the semantics of proofs, definability results are also called full completeness results (starting with [3]), or denotational completeness results [51] (see section 8).

Coming back to compact definability, Milner proved a converse to Proposition 3.3: the fully abstract model of PCF *has to be* (and hence is characterized by the property of being) order-extensional and to enjoy the compact definability property. (A model is called order-extensional when not only the elements of function types are functions, but also their ordering is the pointwise ordering.) The long quest for a fully abstract model started from there.

- The model of stable functions (Berry 77 [12]) was the first to allow to remove some functions such as the "parallel or" that are not definable in the language PCF. The stable model is extensional, but not order-extensional (as functions are ordered by the stable ordering, which is more refined than the pointwise one).
- The model of sequential algorithms [13] was the first model that presented a denotational semantics going beyond the idead of interpretating a program as a black box through which only the input-output behaviour can be observed. Instead, the model of sequential algorithms contains *lror*, *rlor*, `catch` (cf. section 2), etc... . It has been the first "interactive" model of a programming language. This model is not extensional "by design".

More importantly than the lack of (order-)extensionality, compact definability with respect to PCF fails in these two models.

- A famous counterexample witnessing the failure of compact stable functions to be all definable is the Gustave, or Berry-Kleene function f s.t.

$$f(\bot TF) = T, f(T\bot F) = T, f(TF\bot) = T, f(FFF) = F\ .$$

- The counter-examples to compact definability for the model of sequential algorithms in [26, 27] turned out to be influential in the genesis of the AJM game model.

The works on game semantics departed quite radically from this line of work, by renouncing more decisively to extensionality. The HO/AJM game semantics for PCF (restricted to the type hierarchy of PCF) amount to syntax-free descriptions of PCF Böhm trees, which are (possibly infinite) PCF terms in normal form; more precisely, in the HO setting, the interpretation function is a bijection from PCF Böhm trees to innocent and well-bracketed strategies (see section 5).

The game models satisfy thus a fortiori the compact definability property (because the bijection between strategies and PCF Böhm trees restricts to a bijection between finite (= compact) strategies and finite PCF Böhm trees, and because the latter are essentially PCF terms in some canonical form). However, the HO and AJM models fail to be extensional, and they also fail to satisfy the definable separability property.

But there is an "abstract non sense" machinery that forces the definable separability property. One equates any f and g of type A (for all A interpreting a type of the syntax) which cannot be separated by an h, i.e., we set $f =_{op} g$ whenever $hf = hg$ for all h of type $A \to o$. The only delicate point is in verifying that the quotient is still a model (and can in particular interpret the fixpoint construction of PCF).

We stress here that this abstract construction does not exploit any feature of the game model, and as a matter of fact can be applied as well to the model of PCF Böhm trees (which is isomorphic to the HO and AJM models on the finite type hierarchy), yielding a version of Milner's original construction of the fully abstract model of PCF (which was also a quotient of syntax, defined in a slightly different way).

In other words, despite their title, the key contribution of each of the seminal HO and AJM papers is a *definability* result. In some sense, it is the best one could hope for, since it was shown by Loader [80] that the operational equivalence for PCF is not effective (and undecidable for finitary PCF, the version of PCF where the only basic type is the finite type of booleans). This negative result closed the search for a "direct", unquotiented construction of the fully abstract model of PCF, since if such a construction existed, it would surely lead to an effective presentation.

Laird's model of control, that consists of innocent (and not necessarily well-bracketed) strategies, also enjoys definability, but not definable separability, for the same reason as the original HO model: the candidate h for separating f and g is a strategy which is not innocent in general. So, his model needs quotienting, and as a matter of fact Laird has shown how to obtain the model of sequential algorithms as a quotient of his model [73].

Once innocence is removed, then (a variant of) definable separability holds. A variant is needed, as the candidate h is in general not well-bracketed either, but when f and g differ at least in one of their well-bracketed plays, then a well-bracketed h can be found. In the well-bracketed model of Idealized Algol, it holds that $M =_{op} N$ if and only if $[\![M]\!]$ and $[\![N]\!]$ have the same well-bracketed (complete, in the terminology of [6]) plays (see section 6).

The following table summarizes the definability and full abstraction results discussed in this section.

Language	Model	Definability	Full abstraction
PCF + `por`	$Cont$	*Yes*	*Yes*
PCF + `catch`	$SA \approx (\mathbf{G}_{inn} / =_{op})$	*Yes*	*Yes*
PCF	$PCFBT / =_{op}$	*Yes*	*Yes*
PCF	$\mathbf{G}_{AJM}, \mathbf{G}_{HO}, PCFBT$	*Yes*	*No*
PCF + *control*	$\mathbf{G}_{inn}$	*Yes*	*No*
Idealized Algol	$\mathbf{G}_{wb}$	*Yes*	*Yes*

where *Cont*, *SA*, *PCFBT*, $\mathbf{G}_{AJM}$, $\mathbf{G}_{HO}$, $\mathbf{G}_{inn}$, and $\mathbf{G}_{wb}$ stand for the continuous model, the model of sequential algorithms, the PCF Böhm tree model, the AJM model, the HO model, the model of innocent strategies, and (up to a mild quotient) the model of well-bracketed strategies, respectively. (Recall that the three models *PCFBT*, $\mathbf{G}_{AJM}$, $\mathbf{G}_{HO}$ are isomorphic when restricted to the finite type hierarchy.)

On the notion of observables. In the above table, all languages considered share the same notion of observable. We also (not exhaustively) add two other full abstraction results to the list:

- Longley's model of sequentially realizable functionals, or, equivalently, Bucciarelli and Ehrhard's model of strongly stable functions [17] – both these models coincide with the extensional collapse of the model of sequential algorithms – is fully abstract for an extension of PCF with a certain operator H [81].
- Paolini has shown that the stable model is fully abstract for an extension of PCF with some Gustave like operator and a strictness taster which is an extensional version of `catch`.

More generally, when varying the language, one could also think of varying the observable types and values. We mention here only one example, close enough to PCF, and however sharply differing in its properties. Unary PCF is the variant of PCF where the only basic type has only one (non bottom) element, and in which the conditional construct of PCF "degenerates" to a conjunction `if` M `then` N. Loader has shown that the operational equivalence for this language is decidable [79], in sharp constract to the (even finitary) PCF case. As a matter of fact, Laird's model of bistable functions is fully abstract for unary PCF [72].

Around full abstraction The full abstraction problem for PCF has triggered a lot of work, and is to be celebrated for its "side effects".

- The stable model, as reinvented by Girard [44], gave rise to linear logic [45], whose enormous influence in our community should be recounted elsewhere.
- The model of sequential algorithms led me to take cartesian closed categories seriously as a syntax, and from there to the categorical abstract machine [25], and then to explicit substitutions [1].
- The full abstraction problem boosted also the study of logical relations [97, 98, 91], and motivated Bucciarelli and Ehrhard's and Longley's extensional accounts of sequentiality (cf. above).

- Game semantics proved remarkably flexible. Beyond control and first-order references (discussed above), game semantical accounts of a variety of programming features have been given, including subtyping [21], nondeterminism [54], probabilistic choice [36], higher-order references [7], call-by-value [58], or concurrency [43]. In fact, the value of game semantics is precisely here: varying conditions on strategies, we obtain definability results for various languages, and the whole picture establishes a dictionary classifying language constructions in terms of "geometric" properties on strategies.

- Recent (and related) extensions of game semantics concern the possibility of playing several moves together [8], of non necessarily alternating plays [86], of graph (rather than tree) strategies [62, 40, 33], and address the issue of modelling concurrent computation.

- Malacaria and Hankin applied game semantics to program analysis [82, 83, 84]. In a similar vein, applications of game semantics to effective proofs of program equivalences and inequivalences, and more generally to abstract interpretation and model-checking are being developed [42, 92, 37] under the name of *algorithmic game semantics*. We discuss some of these developments in section 7.

Let us mention finally that full abstraction is also of pervasive importance in the theory of process calculi. There, the notion of operational equivalence is bisimulation under many variants, and the notion of model is given by a target calculus: one thus studies fully abstract *translations* of a language into another, in a sense faithful to the original definition of full abstraction.

Some examples of observational equivalence We give examples of pairs of terms M, N that are observationally equivalent in PCF, but are different PCF Böhm trees, and hence not equal in the game models. The term Ω can be considered as an abbreviation of a term such as $Y(\lambda x.x)$ that does never produce any piece of value (i.e., does not have a head normal form, see section 5), or as a special constant that stands for "non-defined", interpreted by $\perp$ in Scott semantics.

$$\left\{\begin{array}{l}\lambda x.\texttt{if}\ x\ \texttt{then}\ M\ \texttt{else}\ N\\ \lambda x.\texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ x\ \texttt{then}\ M\ \texttt{else}\ P)\ \texttt{else}\ N\end{array}\right.$$

$$\left\{\begin{array}{l}\lambda f.\Omega\\ \lambda f.\texttt{if}\ f\Omega\ \texttt{then}\ (\texttt{if}\ fT\ \texttt{then}\ \Omega\ \texttt{else}\ M)\ \texttt{else}\ \Omega\end{array}\right.$$

$$\left\{\begin{array}{l}\lambda f.\Omega\\ \lambda f.\texttt{if}\ f\Omega T\ \texttt{then}\ (\texttt{if}\ fT\Omega\ \texttt{then}\ (\texttt{if}\ fFF\ \texttt{then}\ \Omega\ \texttt{else}\ M)\ \texttt{else}\ \Omega)\ \texttt{else}\ \Omega\end{array}\right.$$

The first pair illustrates a phenomenon of stuttering (cf. section 2). The second one illustrates the fact that PCF is monotonic. The third one illustrates the fact that PCF is sequential. The simplest way to prove $M =_{obs} N$ is to prove $[\![M]\!] = [\![N]\!]$ in a certain model (because $M =_{obs} N$ follows using adequacy). The continuous model suffices for the two frist pairs. The stable model works for the third pair. In this model the function "parallel or" (such that $f\bot T = T$, $fT\bot = T$, and $fFF = F$) does not exist, and therefore the two terms, who could only differ when applied to this function, are not distinguished by the stable model.

We close the section by a a summary of the position of game semantics (and sequential algorithms) with respect to the notions introduced in this section.

1. In section 5, we prove that the strategies respecting two restrictions (innocence, and another, called "well-bracketing") form a model of PCF that satisfies the compact definability property. Actually, the relation between the model and the syntax is much tighter: the interpretation defines a bijective correspondence between terms (in canonical form) and such strategies. This model fails to be fully abstract, and becomes such after a quotient that mimicks the observational equivalence.

2. Moving from PCF to a language called Idealized Algol, and removing the constraint of innocence, we show in section 6 that we get a "true" full abstraction result (without such a quotient).

3. Moving from PCF to a language with control like PCF + `catch`, and removing the constraint of well-bracketing, one obtains a model which needs an observational quotient. On the other hand, the model of sequential algorithms, while lacking the flexibility of the HO approach, provides a "true" full abstraction result for PCF + `catch`, without any quotient.

Exercise 3.1 *The definition of context given above does not force the hole* $[]$ *to occur only once in a context* C*. Write a syntax that enforces this.*

Exercise 3.2 *Justify informally why the function Gustave is not definable in PCF, or (anticipating section 5), why there cannot be a strategy that has this behaviour. Write a couple of terms M, N, based on this function, such that $M \neq_{obs} N$.*

4 Categories of games and strategies

Definition 4.1 *An arena A is given by a set of moves M, which have a polarity O or P (formally, there is function $\lambda_A : M \to \{O, P\}$) and by an enabling relation $\vdash$, which is the disjoint union of a subset of $M \times M$ (one writes $m \vdash n$) and of M (one writes $\vdash m$). If $m \vdash n$, then m and n have opposite polarities, and $\not\vdash n$. If $\vdash m$, then m is an opponent move.*

In section 2, we met two constructions on arenas: product and function space. Here is the general definition.

Definition 4.2 *Let A and B be arenas. The arena $A \times B$ has as moves all m_1 such that m is a move of A and all moves n_2 such that n is a move of B. Polarities of these moves are as in A and B. Enabling is defined as follows:*

$$\frac{\vdash_A m}{\vdash m_1} \quad \frac{\vdash_B n}{\vdash n_2} \quad \frac{m \vdash_A a}{m_1 \vdash a_1} \quad \frac{n \vdash_B b}{n_2 \vdash b_2}$$

Definition 4.3 *Let A and B be arenas. The arena $A \to B$ has as moves all m_1 such that m is a move of A, with polarity opposite to that in A, and all moves n_2 such that n is a move of B, with the same polarity as in B. Enabling is defined as follows:*

$$\frac{\vdash_B n}{\vdash n_2} \quad \frac{\vdash_A m \quad \vdash_B n}{n_2 \vdash m_1} \quad \frac{m \vdash_A a}{m_1 \vdash a_1} \quad \frac{n \vdash_B b}{n_2 \vdash b_2}$$

We adopt a very minimalist view of strategies.

Definition 4.4 *A legal play, or play for short, is a (possibly empty) sequence of moves of alternating polarity which is such that every occurrence of non-initial move is equipped with a pointer to a previous occurrence of a move justifying it. The set of legal plays over an arena A is written L_A. (Clearly, a play starts with Opponent, since only opponent moves can be initial.)*

Definition 4.5 *A strategy on an arena A is a non-empty set of even-length legal plays, which is closed under even-length prefixes.*

Other conditions, like determinism and innocence, will be added later.

The key notion which serves to define the composition of two strategies formally is that of legal interaction. For the moment, we ignore pointers (but we shall use the fact that a play must start with an initial move).

Definition 4.6 *Let A, B, C be three arenas. A legal interaction, or interaction for short, over these arenas is a sequence u of moves from the three arenas such that $u \restriction_{A,B} \in L_{A \to B}$, $u \restriction_{B,C} \in L_{B \to C}$, and $u \restriction_{A,C} \in L_{A \to C}$. We write $int(A, B, C)$ for the set of legal interactions over A, B, C.*

In this definition, say, $u \restriction_{A,B}$ denotes the subsequence of u consisting only of the moves of A, B. One takes care of maintaining the moves of A, B, C all distinct by tagging them if needed.

These three constraints entail that a legal sequence obeys some discipline which one can describe by means of a finite state automaton [53]. We forget about which actual move of arena A, B, or C is played, but only retain from which arena it comes: this makes the alphabet of the automaton. As for the states, they consist of a triple indicating whether it is Opponent's or Player's turn in each of the arenas $A \to B$, $B \to C$, and $A \to C$, respectively. At the beginning, we are thus in state OOO. Now, from state OOO, can we read a move from B? This is impossible because this move would have to be O in both $A \to B$ and $B \to C$, and hence would have to be both O and P as a move of B. But it is not forbidden to read a move of C (which will be O in C), and reach the state OPP (this move does not concern the arena $A \to B$, and hence does not change its O status). Similarly it is not forbidden to read a move of A (which will be P in A), and to reach state POP. Examining the two states OPP and POP, we make similar observations, and moreover we discover no new states, hence our automaton has only three states (other combinations in $\{O, P\}^3$ are not reachable), and is fully descrided as follows:

$$\begin{array}{ll}
OOO \xrightarrow{C} OPP & OOO \xrightarrow{A} POP \\
OPP \xrightarrow{B} POP & OPP \xrightarrow{C} OOO \\
POP \xrightarrow{A} OOO & POP \xrightarrow{B} OPP
\end{array}$$

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

Proposition 4.7 *Let A, B, C be three arenas, and let σ (resp. τ) be a strategy of $A \to B$ (resp. $B \to C$). The following defines a strategy of $A \to C$, called the composition of σ and τ:*

$$\tau \circ \sigma = \{v \mid \exists\, u \in int(A, B, C) \;\; v = u \restriction_{A,C}, u \restriction_{A,B} \in \sigma, v \restriction_{B,C} \in \tau\} \,.$$

(we say that u is a witness of v).

Proof. First we remark that by definition $\tau \circ \sigma$ consists of legal plays. To say that $u \upharpoonright_{A,B}$ is of even length amounts to say that either both $u \upharpoonright_A$ and $u \upharpoonright_B$ are of even length, or both are of odd length. Similarly, $u \upharpoonright_B$ and $u \upharpoonright_C$ are of even length, or both are of odd length. It follows that $u \upharpoonright_A$, $u \upharpoonright_B$, and $u \upharpoonright_C$ either are all of even length, or are all of odd length, and hence a fortiori that $u \upharpoonright_{A,C}$ is of even length. It is obvious that $\tau \circ \sigma$ contains the empty play ϵ. Finally, let $s = u \upharpoonright_{A,C} \in \tau \circ \sigma$ and let s' be an even-length prefix of s. Let u' be a prefix of u such that $s' = u' \upharpoonright_{A,C}$. Since s' ends with a Player's move in $A \to C$, it is Opponent's turn to play in $A \to C$ when u' has been read. But there is only one state with an O at the third position, namely OOO. Hence we also have that $u' \upharpoonright_{A,B}$ and $u' \upharpoonright_{B,C}$ are of even length. But, say, $u' \upharpoonright_{A,B}$ is a prefix of $u \upharpoonright_{A,B} \in \sigma$. Hence, since σ is a strategy, $u' \upharpoonright_{A,B} \in \sigma$. Likewise we have $u' \upharpoonright_{B,C} \in \tau$, from which it follows by definition that $s' \in \tau \circ \sigma$. □

Our next step is to show the associativity of composition. The proof relies on the so-called zipping lemma. The set of interactions $int(A,B,C,D)$ over four arenas is the set of sequences u of moves from the four arenas such that $u \upharpoonright_{A,B} \in L_{A\to B}$, $u \upharpoonright_{B,C} \in L_{B\to C}$, $u \upharpoonright_{C,D} \in L_{C\to D}$, and $u \upharpoonright_{A,D} \in L_{A\to D}$.

Lemma 4.8 *If $u \in int(A,C,D)$ and $v \in int(A,B,C)$ are such that $u \upharpoonright_{A,C} = v \upharpoonright_{A,C}$, then there is a unique $w \in int(A,B,C,D)$ such that $w \upharpoonright_{A,C,D} = u$ and $w \upharpoonright_{A,B,C} = v$.*

Proof. We proceed by induction on the length of $u \upharpoonright_{A,C} = v \upharpoonright_{A,C}$. It it is 0, then a fortiori u has no move from C. But then u has no moves of B either, since $u \upharpoonright_{B,C}$ is legal, and cannot start with a move of B, since no move of B is initial in $B \to C$. Hence $u = \epsilon$, and then we can obviously take $w = v$. Let m be the last occurrence of $u \upharpoonright_{A,C} = v \upharpoonright_{A,C}$, hence u has the form $u'mu''$ with u'' consiting of moves from D only, and similarly v has the form $v'mv''$ with v'' consisting of moves from B only. Suppose that m is an O move in $A \to C$. Then the automaton for arenas A,C,D reading u reaches just after move m a state of the form P??. But there is only one such state, namely POP, from which no moves from D are allowed. This forces $u'' = \epsilon$, i.e., that m is also the last symbol of u. One reasons similarly in the case that m is a P move in $A \to C$, using now the automaton for arenas A,B,C, and we then deduce $v'' = \epsilon$. Thus we have either $u = u'm$ and $v = v'mv''$, or $u = u'mu''$ and $v = v'm$. Suppose, say, that $u = u'm$ and $v = v'mv''$. Then applying induction to u' and v' we obtain w', and we set $w = w'mv''$. (The other case is similar: one sets $w = w'mu''$.). We first check that w is an interaction. We only verify that $w \upharpoonright_{A,B}$ is a legal play (the other cases are similar). Since $w' \upharpoonright_{A,B} = (w' \upharpoonright_{A,B,C}) \upharpoonright_{A,B}$ and $v \in int(A,B,C)$, we have:

$$\begin{array}{rcl} w \upharpoonright_{A,B} & = & (w' \upharpoonright_{A,B})\,((mv'') \upharpoonright_{A,B}) \\ & = & (v' \upharpoonright_{A,B})\,((mv'') \upharpoonright_{A,B}) \quad = \quad v \upharpoonright_{A,B} \quad \in \quad L_{A\to B} \end{array}$$

Next we check that w satisfies the specification in the statement:

$$\begin{array}{l} w \upharpoonright_{A,C,D} = (w' \upharpoonright_{A,C,D})m = u'm = u \\ w \upharpoonright_{A,B,C} = (w' \upharpoonright_{A,B,C})mv'' = v \end{array}$$

Finally, we prove that w is unique. Since $w \upharpoonright_{A,C,D} = u'm$, we can write $w = w'mw''$ with w'' consisting of moves from B only. Let us compute $w' \upharpoonright_{A,C,D}$ and $w' \upharpoonright_{A,B,C}$. From $w \upharpoonright_{A,C,D} = u$ and $w \upharpoonright_{A,C,D} = (w' \upharpoonright_{A,C,D})m$, we get immediately $w' \upharpoonright_{A,C,D} = u'$. From $w \upharpoonright_{A,B,C} = v$ and $w \upharpoonright_{A,B,C} = (w' \upharpoonright_{A,B,C})mw''$ we get the equality $v'mv'' = (w' \upharpoonright_{A,B,C})mw''$. We notice that v'' cannot be a strict prefix of w'' (nor vice-versa) since this would force m to be a move from B, but m is a move of $A \to C$ (note here that we use an assumption that A,B, and C are disjoint arenas – we shall come back to this when we shall define the identities). Hence $w'' = v''$ and $w' \upharpoonright_{A,B,C} = v'$. But we know by induction that w' is unique. Moreover, we have also proved that w'' is determined (and equal to v''). Hence w is completely determined. □

Proposition 4.9 *If σ, τ, υ are strategies of $A \to B, B \to C$, and $C \to D$, respectively, then $\upsilon \circ (\tau \circ \sigma) = (\upsilon \circ \tau) \circ \sigma$.*

PROOF. Let $s \in \upsilon \circ (\tau \circ \sigma)$, and let u be a witness of s, i.e., $u \in \mathit{int}(A, C, D)$, $s = u \upharpoonright_{A,D}$, $u \upharpoonright_{C,D} \in \upsilon$, and $u \upharpoonright_{A,C} \in \tau \circ \sigma$. Let v be a witness of $u \upharpoonright_{A,C}$, i.e., $v \in \mathit{int}(A, B, C)$, $u \upharpoonright_{A,C} = v \upharpoonright_{A,C}$, $v \upharpoonright_{A,B} \in \sigma$, and $v \upharpoonright_{B,C} \in \tau$. By Lemma 4.8, we get $w \in \mathit{int}(A, B, C, D)$ such that $w \upharpoonright_{A,C,D} = u$ and $w \upharpoonright_{A,B,C} = v$. We claim that $w \upharpoonright_{A,B,D} = u'$ is a witness of s as a member of $(\upsilon \circ \tau) \circ \sigma$. We have indeed: $u' \upharpoonright_{A,D} = w \upharpoonright_{A,D} = u \upharpoonright_{A,D} = s$ and $u' \upharpoonright_{A,B} = v \upharpoonright_{A,B} \in \sigma$, and we are left to check that $u' \upharpoonright_{B,D} \in \upsilon \circ \tau$. This follows from the fact that $w \upharpoonright_{B,C,D}$ is a witness of $u' \upharpoonright_{B,D}$:

$$\begin{array}{l} (w \upharpoonright_{B,C,D}) \upharpoonright_{B,D} = w \upharpoonright_{B,D} = u' \upharpoonright_{B,D} \\ w \upharpoonright_{B,C} = v \upharpoonright_{B,C} \in \tau \\ w \upharpoonright_{C,D} = u \upharpoonright_{C,D} \in \upsilon \end{array}$$

We have thus proved that $\upsilon \circ (\tau \circ \sigma) \subseteq (\upsilon \circ \tau) \circ \sigma$. The converse inclusion is proved similarly. □

Let us remark that the uniqueness of w in Lemma 4.8 has not served here.

We next define the identity strategies, which are often called copy-cat strategies. We first meet a delicate point, namely that we deal with the arena $A \to A$. How can we make sure that a move comes from the left A or the right A? This difficulty actually appears above too. When defining the composition of two strategies σ of $A \to B$ and τ of $B \to C$, we have used explicitly in our proofs that the moves from the three arenas are disjoint. This difficulty can be (and must be) solved by temporarily "fattening" the moves of the relevant arenas to ensure their disjointness. (Note that we have done this all the way through in section 2.) Here is a more correct definition of composition. Given three arenas A, B, C and strategies σ, τ of $A \to B$ and $B \to C$, respectively, take the three arenas A_1, B_2, C_3, where, say, A_1 is the arena obtained from A by transforming each move a to a_1. Take σ' and τ' be the obvious renamings of σ and τ, apply the above definition to σ' and τ', and then perform the obvious inverse renaming on the composition. But wait, this is not yet quite right, because even the definition of σ or τ may involve a need for renaming. Our constructions of the arena $A \to B$ is based on a disjoint union of (the moves of) A and B. Hence σ itself is given in fact through a prior tagging of A and B, say, A_α and B_β (with $\alpha \neq \beta$ – similar assumptions are implicit below). Similarly, τ is given through taggings B_γ and C_δ. The procedure is to rename σ and τ through any tagging, say A_λ, B_μ, and C_ν (in such a way that the two occurrences of B now have the same tagging), then to compose, and then to rename to whatever tagging of A, C is needed. The important point of this construction is that if we compose in this way, then the result does not depend on the choice of the intermediate tags λ, μ, ν. Morever, the following also yields identical results:

- to express the composition with respect to, say, A_σ and C_τ and then rename it according to, say, $A_{\sigma'}$ and $C_{\tau'}$,
- or to express it directly with respect to $A_{\sigma'}$ and $C_{\tau'}$.

We can come back to the identity strategies. We are looking for a morphism from, say, A_1 to A_2.

Proposition 4.10 *Let A be an arena. We are considering words u over the alphabet $A_1 \cup A_2$. We write, say, $u \upharpoonright_1$ for the result of removing the tags $_1$ in $u \upharpoonright_{A_1}$. We define*

$$id' = \{u \in L_{A \to A} \mid v \upharpoonright_1 = v \upharpoonright_2 \text{ for all even prefixes } v \text{ of } u\}$$

We define id'' as the smallest set of plays closed under the following rules:

$$\frac{}{\epsilon \in id''} \qquad \frac{v \in id'', a\ O\ move}{va_2a_1 \in id''} \qquad \frac{v \in id'', a\ P\ move}{va_1a_2 \in id''}$$

We have $id' = id''$ (id for short), and id is a strategy.

Proof. We note that id'' is a strategy by construction (the plays are legal by the assumptions on a). It is obvious by induction on the definition of id'' that $id'' \subseteq id$. We prove the converse inclusion by induction on the length of $u \in id'$. First we note that u is of even length, since $u \upharpoonright_1 = u \upharpoonright_2$. We can thus write $u = vmn$, and we can apply induction to v, which belongs to id' by definition of id'. Hence $v \in id''$. On the other hand, the constraints $u \upharpoonright_1 = u \upharpoonright_2$ and $v \upharpoonright_1 = v \upharpoonright_2$ entail that m and n have the same underlying move of A, i.e., either $m = a_2$ and $n = a_1$, or $m = a_1$ and $n = a_2$. In the first (resp. second) case, because u is legal, then a must be an O move (resp. P move). It follows that $u \in id''$. $\square$

For example, the identity morphism $id = \mathtt{comm}_1 \to \mathtt{comm}_\epsilon$ consists of all the finite even prefixes of the infinite word

$$((\mathtt{run}_\epsilon\mathtt{run}_1)^+(\mathtt{done}_1\mathtt{done}_\epsilon)^+)^\infty \ .$$

As another instance, $id : (\mathtt{comm}_{11} \to \mathtt{comm}_1) \to (\mathtt{comm}_2 \to \mathtt{comm}_\epsilon)$ contains the play

$$\mathtt{run}_\epsilon\mathtt{run}_1\mathtt{run}_{11}\mathtt{run}_2\mathtt{done}_2\mathtt{done}_{11}\mathtt{done}_1\mathtt{done}_\epsilon$$

(this word is obtained by merging $\mathtt{run}_\epsilon\mathtt{run}_2\mathtt{done}_2\mathtt{done}_\epsilon$ and $\mathtt{run}_1\mathtt{run}_{11}\mathtt{done}_{11}\mathtt{done}_1$).

We now show that the identity is indeed an identity morphism. The following lemma gives us some inductive handle.

Lemma 4.11 *Let A, B, C be three arenas and let $u \in int(A, B, C)$. Then u is a sequence of blocks of the form $mb_1 \dots b_kn$ where m is an O $(A \to C)$ move, the b_i's are B moves, and n is a P $(A \to C)$ move. Moreover, if u is a witness for $\tau \circ \sigma$ (i.e., $u \upharpoonright_{A,B} \in \sigma$, $u \upharpoonright_{B,C} \in \tau$), and if $u = u'mb_1 \dots b_kn$ where $mb_1 \dots b_kn$ is as above, then u' is also a witness.*

Proof. We first notice that because $u \upharpoonright_{A,C}$ is legal, u has the form $\dots m_1 \dots m_2 \dots m_3 \dots$, where the $\dots$ are B moves and the m_i's are $A \to C$ moves, where m_1 is an O move, m_2 a P move, m_3 an O move, etc... Just after a P $(A \to C)$ move, the automaton must be in the state OOO, from which it follows that no B move is allowed. Hence the $\dots$ between m_2 and m_3 (and more generally between m_{2k} and m_{2k+1}) are empty. The initial $\dots$ before m_1 must be empty, since the B moves are not initial in $B \to C$. The first part of the statement follows easily. For the second part, we just have to check that $u' \upharpoonright_{A,B}$ and $u' \upharpoonright_{B,C}$ are of even length, and this follows readily from the fact that just before reading m, the automaton is in state OOO and from the fact that $u' \upharpoonright_{A,B}$ and $u' \upharpoonright_{B,C}$ are legal. $\square$

Proposition 4.12 *We have always $id \circ \sigma = \sigma$ and $\sigma \circ id = \sigma$.*

Proof. We prove only the first equality (the second is proved similarly). We consider that id goes from B_1 to B_2. Let $v \in \sigma$, considered as going from A to B_2. We shall show $v \in id \circ \sigma$. by exhibiting a witness u. We proceed by induction on the length of v Let $v = v'mn$, and let u' be a witness for v'. To be precise, our induction load (L) is:

$$u' \upharpoonright_{A,B_2} = v' \qquad u' \upharpoonright_{B_1,B_2} \in id \qquad u' \upharpoonright_{A,B_1} = u' \upharpoonright_{A,B_2}$$

There are four cases:

1. m is P A and $n = b_2$ is P B_2. Then take $u = u'mb_1b_2$.
2. m is P A and n is O A. Then take $u = u'mn$.
3. $m = b_2'$ is O B_2 and $n = b_2$ is P B_2. Then take $u = u'b_2'b_1'b_1b_2$.
4. $m = b_2'$ is O B_2 and n is O A. Then take $u = u'b_2'b_1'n$.

The definition is tailored to ensure that u satisfies (L). As a consequence we have: $u \restriction_{A,B_1} = u \restriction_{A,B_2} \in \sigma$, hence u is a witness.

We next show that every witness u is such that $u \restriction_{A,B_1} = u \restriction_{A,B_2}$. The converse inclusion will follow immediately, since if $v \in id \circ \sigma$, then $v = u \restriction_{A,B_2}$ for some witness u, and we have $v \in \sigma$ since $u \restriction_{A,B_1} \in \sigma$ and $u \restriction_{A,B_1} = u \restriction_{A,B_2}$. We proceed by induction, our unit being the blocks identified in Lemma 4.11. Let $u = u'mb_1^1 \ldots b_1^k n$ $(k \geq 0)$ where m is O $A \to B_2$, n is P $A \to B_2$, and the b^i's are B moves. There are two cases:

1. $m = b_2$ is an O B_2 move. Then since $u' \restriction_{B_1,B_2} \in id$, and by the definition id'' of id (cf. Definition 4.10), we have $k \geq 1$ and $b^1 = b$. There are two subcases:
   (a) $k = 1$, i.e., $u = u'mb_1^1 n$. After b_1^1, the automaton is in a state $?OP$, which must be POP, out of which no B_2 move is allowed. Hence n is an O A move. It follows that $u \restriction_{A,B_1} = u' \restriction_{A,B_1} b_1 n = u' \restriction_{A,B_2} b_2 n = u \restriction_{A,B_2}$.
   (b) $k > 1$. By definition of id, b^2 is P B, and b_1^2 must be followed by b_2^2. It follows that $k = 2$ and $n = b_2^2$. Altogether we have $u = u'b^2b^1b_1^2b_2^2$, and the conclusion follows immediately.
2. m is P A. There are again two subcases:
   (a) $k = 0$. Then $u = u'mn$ and after m the automaton is in state POP. This forces again n to be in A. Then $u \restriction_{A,B_1} = u' \restriction_{A,B_1} mn = u' \restriction_{A,B_2} mn = u \restriction_{A,B_2}$.
   (b) $k > 0$. Then by definition of id $(b_1, 1)$ must be P B_1 and must be followed by $(b_1, 2)$ in $u \restriction_{B_1,B_2}$, which forces $k = 1$ and $n = (b_1, 2)$, i.e., $u = u'm(b_1, 1)(b_1, 2)$ and the conclusion follows immediately. □

Altogether, we have obtained a category. Next, when we shall impose a condition on strategies, we shall make sure that it is stable under composition and that the copy-cat strategy respects it, so as to obtain a subcategory.

Definition 4.13 *A strategy σ is called* deterministic *when the following implication holds, for all s, m, n_1, n_2:*

$$smn_1, smn_2 \in \sigma \;\Rightarrow\; n_1 = n_2$$

Definition 4.14 *The P view $\ulcorner s \urcorner$ of a play s is defined as follows:*

$$\begin{array}{ll}
\ulcorner \epsilon \urcorner = \epsilon & \\
\ulcorner sn \urcorner = \ulcorner s \urcorner n & (n \textit{ P move}) \\
\ulcorner sm \urcorner = m & (m \textit{ initial}) \\
\ulcorner sns'm \urcorner = \ulcorner sn \urcorner m & (m \textit{ O move}, m \textit{ points to } n)
\end{array}$$

Definition 4.15 *A deterministic strategy is called innocent if the following equivalence holds, for all s:*

$$s \in \sigma \Leftrightarrow \ulcorner s \urcorner \in \sigma$$

Theorem 4.16 *Arenas and strategies form a category. It contains as subcategories the categories of arenas and deterministic strategies, and of arenas and innocent strategies.*

Fat and meager. We have thus defined composition "mathematically" (or "denotationally"), i.e., without reference to a computing device (although such a device is immediately behind, in the form of the automaton that we have described). How does this compare with the operational definitions of composition that have been explained on examples in section 2? It can be shown that, *for innocent strategies*, the *view function* (i.e., the set of views) of the composition of two strategies as defined in this section coincides with the view function obtained by composing the view functions of the two strategies, using the View Abstract Machine or the Geometric Abstract Machine. The fat representation is well-suited for a quick and synthetic definition of composition, while the meager representation has a closer fit to syntax, including in operational aspects.

Exercise 4.1 *Prove the second part of Theorem 4.16.*

Exercise 4.2 *Check that all the examples given in section 2 are innocent strategies.*

5 PCF Böhm trees and their HO characterization

Convention. We observe that any PCF type can be written as $A^1 \to \ldots \to A^n \to \texttt{base}$ (where `base` is `nat` or `bool`). We write PCF types as trees, and we individuate every node by the path that connects it to the root, and that is codified by a word (the empty word ϵ for the root, the word 1 for the first son, etc...) For example, $(\texttt{bool}_{11} \to \texttt{bool}_1) \to \texttt{bool}_2 \to \texttt{bool}_\epsilon$, or, as a tree:

$$\texttt{bool}_\epsilon \left\{ \begin{array}{l} \texttt{bool}_1 \left\{ \texttt{bool}_{11} \right. \\ \texttt{bool}_2 \end{array} \right.$$

We want to bring closer to each other the syntax of PCF on one hand and the strategies on PCF arenas on the other hand, to the point of getting a bijective correspondence:

1. We add a constant Ω at all types to the syntax of PCF (cf. section 3):

$$M ::= \ldots \mid \Omega$$

   The constant Ω is a bit like [], with the additional idea that the hole waits for a value that has not yet been computed (and that may never arrive, if the computation does not converge). We take only the terms in (η-long, see below) normal form of M of PCF (see Exercise 5.2):

   (a) The normal forms are possibly infinite (in depth): this is due to the presence of Y, that allows to program any recursive function (see Exercise 5.1).

   (b) If M has type $A^1 \to \ldots \to A^n \to \texttt{base}$, we compute the normal form, not of M, but of its so-called *η-long* form

$$\lambda x_1 \ldots x_n.\texttt{case}\ (Mx_1 \ldots x_n)\ [\ldots, c \to c, \ldots]$$

   We observe that if $\texttt{base} = \texttt{nat}$, we now also have trees that are infinite in width.

   We have seen through examples (cf. section 2) that any such term, called *PCF Böhm tree*, can be read directly as a strategy (more precisely, as a view function): the transformation is textual and one-to-one.

2. To turn the correspondence into a bijection, it remains to characterize the image of this transformation. It consists in the strategies that satisfy two conditions (see below).

PCF Böhm trees are made of elementary bricks, called **head normal forms**:

$$\begin{array}{l} M = \lambda x_1 \ldots x_n.W \\ W = \texttt{case}\ xM_1 \ldots M_p\ [\ldots, v_i \to W_i, \ldots]\ . \end{array}$$

One should look at $\lambda x_1 \ldots x_n$ as an Opponent move, and to $\texttt{case}\ x$ as a Player move. The idea is that in a normal form of PCF, all the configurations $(\lambda x.M)N$ and all the occurrences of Y (by repeated unfolding of the recursive definitions, using the rules $YM = M(YM)$) disappear; the other constants except n, T, F also disappear (think that $succ5 = 6$, etc...).

A **PCF Böhm tree** is a term that is hereditarily in head normal form:

$$\lambda x_1 \ldots x_n.\texttt{case}\ x \begin{cases} M_1 \\ \vdots \\ \lambda y_1 \ldots y_p.\texttt{case}\ y \begin{cases} N_1 \\ \vdots \\ N_q \\ \vdots \\ w \to \ldots \\ \vdots \end{cases} \\ \vdots \\ M_p \\ \vdots \\ v \to \texttt{case}\ z \begin{cases} \vdots \end{cases} \\ \vdots \end{cases}$$

We illustrate on *lor*, of type $\texttt{Bool}_1 \to \texttt{Bool}_2 \to \texttt{Bool}_\epsilon$ (cf. section 2), how a PCF Böhm tree can be viewed as a strategy:

$$\left\{ \lambda xy.\texttt{case}\ x \begin{cases} T \to T \\ F \to \texttt{case}\ y \begin{cases} T \to T \\ F \to F \end{cases} \end{cases} \right. \qquad \left\{ q_\epsilon\, q_1 \begin{cases} T_1\, T_\epsilon \\ F_1\, q_2 \begin{cases} T_2\, T_\epsilon \\ F_2\, F_\epsilon \end{cases} \end{cases} \right.$$

Here are the two conditions characterizing the interpretations of PCF Böhm trees:

1. **Innocence**. Cf. Definition 4.15. A way of looking at innocence is to say that a strategy contains only "views". In a view, every move of the opponent must point to the immediately preceding move. In particular, an initial move of the arena can occur only as the first move in a view.

2. **"Well-bracketing"**. We observe that the arenas of PCF are made of copies, or occurrences, of the base arenas. All the moves that are occurrences of q are called questions, and all the moves that are occurrences of n, T, F are called answers. A view (more generally a play) is called "well-bracketed" when every answer points to the last open question. The strategy `catch` does not respect the condition of well-bracketing, because it contains the view $q\, q_1\, q_{11}\, T$, where the T answers to q while q_1 and q_{11} are still open.

Theorem 5.1 *Every innocent and well-bracketed strategy σ that belongs to the arena interpreting some type $A = A^1 \to A^2 \to \ldots \to A^n \to \mathtt{base}_\epsilon$ of PCF, considered as a set of views, can be read as (is the interpretation of) a term of PCF (in normal form).*

Proof. • σ contains only the empty view. Then $\sigma = [\![\Omega]\!]$.

• σ contains the view $q_\epsilon\, v_\epsilon$. Then $\sigma = [\![\lambda x_1 \ldots x_n.v]\!]$. Indeed, we have $[\![\lambda x_1 \ldots x_n.v]\!] = \{\epsilon, q\, v\}$ on one hand, while, on the other hand, if $q\, v \in \sigma$, then σ cannot contain other views, because a move that would follow v should be justified by v, by the innocence condition, and there is no such move (easily proved by induction on PCF types).

• σ contains the view $q_\epsilon\, q_i$. If $A^i = B^{i1} \to \ldots \to B^{im} \to \mathtt{base}_i$, we wish to write

$$\sigma = [\![\lambda x_1 \ldots x_n.\mathtt{case}\, x_i M^1 \ldots M^m\ [\ldots, v \to W, \ldots]]\!]$$

where every $\lambda x_1 \ldots x_n.M^j$ belongs to (the interpretation of) $A^1 \to \ldots \to A^n \to B^{ij}$ and $\lambda x_1 \ldots x_n.W$ belongs to A. Indeed, we obtain strategies σ^j and τ with these types, as follows:

$$\sigma^j = \{q_{\epsilon'} s' \mid q_\epsilon q_i q_{ij} s \in \sigma\}$$
$$\tau = \{q_\epsilon s \mid q_\epsilon q_i v s \in \sigma\}$$

The definition of σ^j takes into account the fact that in the type $A^1 \to \ldots \to A^n \to B^{ij}$, $B^{ij} = C^{ij1} \to \ldots C^{ijl} \to \mathtt{base}_{ij}$ still occurs as subtype of A^i, but now also occurs as the type to the right of A^n (in terms of linear logic, we are taking a copy of the type B^{ij}, and this corresponds to the equation $!B^{ij} = B^{ij} \otimes (!B^{ij})$). Precisely, s' is s, modified as follows:

- every move n_{ij} pointing to the first occurrence of q_{ij} in $q_\epsilon q_i q_{ij} s$ becomes $n_{\epsilon'}$ and points to $q_{\epsilon'}$;
- every move m_{ijlu} hereditarily justified by the first occurrence of q_{ij} in $q_\epsilon q_i q_{ij} s$ becomes $m_{(n+l)u}$, and m_{n+l} is made to point to $q_{\epsilon'}$;
- all the question moves that were pointing to q_ϵ in s are made to point now to $q_{\epsilon'}$ in $q_{\epsilon'} s'$;
- all the other moves remain the same.

This transformation is "well-typed", since, in $q_{\epsilon'} s'$, $q_{\epsilon'}$ is the initial move in the arena

$$A^1 \to \ldots \to A^n \to B^{ij} = A^1 \to \ldots \to A^n \to C^{n+1} \to \ldots C^{n+l} \to \mathtt{base}_{\epsilon'}\ ,$$

where we have renamed $C^{ij1}, \ldots C^{ijl}, \mathtt{base}_{ij}$ as $C^{n+1}, \ldots, C^{n+l}, \mathtt{base}_{\epsilon'}$. Hence $q_{\epsilon'}$ enables moves q_i $(i \leq n)$ as well as moves q_{n+l} $(l \leq k)$. It may be helpful to look at the respective tree structures of A and of $A^1 \to \ldots \to A^n \to B^{ij}$:

$$\mathtt{base}_\epsilon \left\{ \begin{array}{l} A^1 \\ \vdots \\ \mathtt{base}_i \left\{ \begin{array}{l} B^{i1} \\ \vdots \\ \mathtt{base}_{ij} \left\{ \begin{array}{l} C^{ij1} \\ \vdots \\ C^{ijl} \end{array} \right. \\ \vdots \\ B^{im} \end{array} \right. \\ \vdots \\ A^n \end{array} \right. \qquad \mathtt{base}_{\epsilon'} \left\{ \begin{array}{l} A^1 \\ \vdots \\ \mathtt{base}_i \left\{ \begin{array}{l} B^{i1} \\ \vdots \\ \mathtt{base}_{ij} \left\{ \begin{array}{l} C^{ij1} \\ \vdots \\ C^{ijl} \end{array} \right. \\ \vdots \\ B^{im} \end{array} \right. \\ \vdots \\ A^n \\ C^{n+1} \\ \vdots \\ C^{n+l} \end{array} \right.$$

The redirected pointers originate from moves of the first kind, while the moves of the second kind (and the moves they justify) are the renamed ones.

Then we can find terms $\lambda x_1 \ldots x_n.M^j$ and $\lambda x_1 \ldots x_n.W$ whose interpretation are σ^j and τ, respectively, by induction on the size of σ, i.e., the number of nodes in the tree representation of σ (In the infinite case, one reasons by coinduction.)

We are left to show that σ^j and τ are innocent and well-bracketed strategies, of the right type. In the following arguments we use the fact that σ contains only views.

- σ^j. Consider a view $q_{\epsilon'} s' \in \sigma^j$. We have to prove that it is safe to replace the prefix $q_\epsilon q_i q_{ij}$ of $q_\epsilon q_i q_{ij} s$ by the prefix $q_{\epsilon'}$ of s', which given the definition of s' amount to show that no move of s points to q_i and that no *answer* move v of s points to q_ϵ. By definition of a view, a move justified by q_i, being an O move, can only be the move immediately following q_i in $q_\epsilon q_i q_{ij} s$, namely q_{ij}, which establishes the first property. As for the second, suppose that a move v of s points (is the answer) to the first move q_ϵ. Then, by well-bracketing, there should be a move v' that answers the question q_i. But, as we have seen, q_{ij} is the only move pointing to q_i, and it is a question.

- τ. Here we have to convince ourselves that we can cut the subsequence $q_i v$. It is enough to prove that no move of s points to q_i nor to v.

  - q_i. If w in $q_\epsilon q_i v s$ points to q_i, then, by definition of a view, it must follow q_i immediately.

– v. As noted above, an answer never justifies another move. □

Exercise 5.1 *(Example of a Böhm tree of infinite depth.) Compute the infinite normal form of* $Y(\lambda fxy.x(fy)$.

Exercise 5.2 *Figure out which rewriting rules must be added to those given above in order to achieve that starting from any term M of PCF and reducing*

$$\lambda x_1 \dots x_n.\texttt{case}\ (Mx_1 \dots x_n)\ [\dots, c \to c]$$

with a call-by-name strategy, i.e., always reducing the leftmost redex, the computation is not blocked before reaching a head normal form as above. For example,

$$\lambda x.\texttt{case}\ (\texttt{succ}\ x)\ [\dots, v \to v, \dots]$$

is in normal form (according to the rules of section 3) but is not of the form

$$\lambda x.\texttt{case}\ x\ [\dots, v \to v, \dots]$$

hence one should add a rule, etc...

Exercise 5.3 *(On the diffference play/view.) Enumerate all the possible views of type* $\texttt{Bool}_1 \times \texttt{Bool}_2$ *(cf. section 2). Express the couple (F,T) as a set σ of views, and also as a strategy σ' with all the possible plays (i.e., $\sigma' = \{s \mid \ulcorner s \urcorner \in \sigma\}$). Which play corresponds to the interaction of lor with (F,T)?*

Exercise 5.4 *Give examples of sequences of moves relative to the arena* $(\texttt{bool}_{11} \to \texttt{bool}_1) \to \texttt{bool}_2 \to \texttt{bool}_\epsilon$ *that are not alternating, not justified, are not a view, are not well-bracketed, respectively. Enumerate all the well-bracketed views of this type, and all the innocent and well-bracketed strategies, and verify Theorem 5.1 on this type.*

Exercise 5.5 *Describe the interaction between the term / strategy $\lambda f.f(fT\Omega)(f\Omega T)$ and the strategy lor and with the strategy ror.*

Exercise 5.6 *Consider now a base type* `unit` *with only one constant $*$. Define a variant of the strategy* `catch` *of type* $(\texttt{unit} \to \texttt{unit} \to \texttt{unit}) \to \texttt{nat}$ *and a strategy / term of the form* $\lambda f.\texttt{case}\ \texttt{catch}(f)\ [0 \to \dots, 1 \to \dots, 2 \to \dots]$*, where the terms $\dots$ do not have nested occurrences of f, behaving in the same way as $\lambda f.f(f * \Omega)(f\Omega *)$, in the sense that the two strategies applied to every strategy produce the same result.*

Exercise 5.7 *In this exercise (which follows [77]), we limit ourselves to the λ-calculus (no $n, T, F,$ `case`), typed with only one base type, let us call it* `q` *(without even constants of this base type). Prove that the normal forms (Böhm trees), i.e., the terms of type $A_1 \to \ldots \to A_n \to \mathtt{q}$ that have (hereditarily) the (η-long) form $\lambda x_1 \ldots x_n.xM_1 \ldots M_p$, are in bijective correspondence with the innocent strategies of the arena $A_1 \to \ldots \to A_n \to \mathtt{q}$, interpreting* `q` *as the arena having q as unique move. (In other words, the distinction between questions and answers, and the well-bracketing condition, concern PCF, not the underlying λ-calculus.)*

6 Imperative features

In this section (based on [6, 5], but see also [95]), we model commands (or instructions) and cells. The arena `comm` has only two moves: `run` and `done`, that are respectively O and P. In imperative mode, all the computation occurs as side effects (memory updates), hence the unique thing that matters for a command is when it is initiated (`run`) and when it ends (`done`). The sequencing of two commands is modelled by the following strategy of $\mathtt{comm}_1 \times \mathtt{comm}_2 \to \mathtt{comm}_\epsilon$:

$$\mathtt{run}_\epsilon\ \mathtt{run}_1\ \mathtt{done}_1\ \mathtt{run}_2\ \mathtt{done}_2\ \mathtt{done}_\epsilon$$

whose role is to initiate its first argument, then to wait for the end signal of this argument, in order to initiate the second argument, and finally to give the hand back (to the context) when the second argument has finished its job.

Now we model a cell. The arena `var` (or `nat ref` in a language like ML) has the following moves:

O moves	`read` $\mathtt{write}(n)$ $(n \in \omega)$
P moves	`OK` n $(n \in \omega)$

The move `OK` signals that a writing request has been satisfied, while n is the current value that is given in responce to a reading request. The dependencies are $\mathtt{write}(n) \vdash \mathtt{OK}$ and $\mathtt{read} \vdash n$ $(n \in \omega)$. We define the strategy `cell` as the smallest strategy closed under the following rules:

$$\frac{}{\epsilon \in \mathtt{cell}} \qquad \frac{u \in \mathtt{cell}}{u\,\mathtt{write}(n)\,\mathtt{OK} \in \mathtt{cell}} \qquad \frac{u \in \mathtt{cell} \quad \mathtt{cv}(u) = m}{u\,\mathtt{read}\,m \in \mathtt{cell}}$$

where `cv` (current value) is (partially) defined by the following rules:

$$\frac{\mathtt{cv}(u) \text{ defined}}{\mathtt{cv}(u\,\mathtt{read}\,m \in \mathtt{cell}) = \mathtt{cv}(u)} \qquad \frac{}{\mathtt{cv}(u\,\mathtt{write}(m)\,\mathtt{OK}) = m}$$

In particular, `cv` is not defined on the empty play ϵ, hence $\texttt{read}\, m$ is not a play of `cell`. This is because a cell cannot satisfy a request `read` until at least one `write` has been performed. But the most important thing is that the rest of the specification of `cell` says that a a request `read` is always answered with the last written value.

Then comes an important observation: the strategy `cell` *is not innocent.* For example, it contains $\texttt{write}(0)\ \texttt{OK}\,\texttt{read}\, 0$ and $\texttt{write}(2)\ \texttt{OK}\,\texttt{read}\, 2$, but

$$\texttt{write}(0)\ \texttt{OK}\,\texttt{read} \quad \text{and} \quad \texttt{write}(2)\ \texttt{OK}\,\texttt{read}$$

have the same view, that is `read` (remember that `read` is an initial move).

Now we give the semantics of assignment, say $x := 2$. We interpret this command as the following strategy of type $\texttt{var}_1 \to \texttt{comm}_\epsilon$ (because it depends on a free parameter x of type `var`):

$$\texttt{run}_\epsilon\ \texttt{write}(2)_1\ \texttt{OK}_1\ \texttt{done}_\epsilon$$

A slightly more complicated example is $x := y + 2$, whose interpretation is the following strategy of $\texttt{var}_1 \times \texttt{var}_2 \to \texttt{comm}_\epsilon$:

$$\texttt{run}_\epsilon\ \texttt{read}_2 \begin{cases} \vdots \\ n_2\ \texttt{write}(n+2)_1\ \texttt{OK}_1\ \texttt{done}_\epsilon \\ \vdots \end{cases}$$

But we have been too rapid. There is typing problem in $y + 2$, because y is of type `var`, not of type `nat`. We should consider an explicit coertion $!y$, where ! (dereferencing) has type $\texttt{var}_1 \to \texttt{nat}_\epsilon$ and is interpreted by the strategy that contains

$$\begin{array}{l} \epsilon \\ q_\epsilon\ \texttt{read}_1\ n_1\ n_\epsilon \\ q_\epsilon\ \texttt{read}_1\ n_1\ n_\epsilon\ q_\epsilon\ \texttt{read}_1\ p_1\ p_\epsilon \\ \vdots \end{array}$$

The strategy interpreting x ::=$!y + 2$ is obtained by composing a strategy ::= of type $\texttt{var} \times \texttt{nat} \to \texttt{comm}$ with the strategy $id \times ((\_ + 2)\circ!)$ of type $(\texttt{var} \times \texttt{var}) \to (\texttt{var} \times \texttt{nat})$, where $(\_ + 2)$ is the strategy $[\![\lambda x.x + 2]\!]$ (see Exercise 6.2).

Then we combine sequencing and assignment. Here is the interpretation of

$$(x := 0); (x := x + 1)$$

as a strategy of $\texttt{var}_1 \to \texttt{comm}_\epsilon$:

$$\texttt{run}_\epsilon\ \texttt{write}(0)_1\ \texttt{OK}_1\ \texttt{read}_1 \begin{cases} \vdots \\ n_1\ \texttt{write}(n+1)_1\ \texttt{OK}_1\ \texttt{done}_\epsilon \\ \vdots \end{cases}$$

We observe that the interpretation of $(x := 0); (x := x + 1)$ does not respect the discipline of a cell. The move n_1 can be arbitrary, it is not forced to be 0. In fact, this command could be inserted in a context causing an interference. For example, it could be part of a distributed system sharing the resource x, as in

$$((x := 0); (x := x + 1)) \parallel (x := 2$$

where $\parallel$ places its two arguments in parallel. Since the two arguments share the global variable x, they are not independent. The "atomic" (or unbreakable) pieces of program that have to access x are $(x := 0)$, $(x := x + 1)$ and $(x := 2)$, call them P_1, P_2, and P_3, respectively. Atomic means that while executed, no other piece of program can interfer with its access to x (either for reading or for writing). A possibility is that P_3 be scheduled after P_1 and before P_2, in which case, executing $x := x + 1$, the value read is not 0 but 2. Hence the above "liberal" semantics for $(x := 0); (x := x + 1)$ is appropriate.

But a programmer may want to enforce the atomicity of the whole command $f = ((x := 0); (x := x + 1))$. This effect can be obtained using a "block declaration", that makes x local to f. Here is an ALGOL-like syntax for such a declaration:

$$\texttt{new}\, x.((x := 0); (x := x + 1)) \ .$$

In this program Q, x is bound. Hence Q must be interpreted, not as a strategy of $\texttt{var} \to \texttt{comm}$, but as a strategy of $\texttt{comm}$, obtained by applying f to $\texttt{cell}$. But such a strategy can only be $\texttt{run}\,\texttt{done}$. Frustrating? No, because what is interesting is the interaction (hiding = side effect!). In fact it can be verified that the interaction of $f = ((x := 0); (x := x+1))$ with $\texttt{cell}$ results in the following play:

$$\texttt{run}_\epsilon\ \texttt{write}(0)_1\ \texttt{OK}_1\ \texttt{read}_1\ 0_1\ \texttt{write}(1)_1\ \texttt{OK}_1\ \texttt{done}_\epsilon$$

In particular, the interaction forces a path in f that obeys the cell discipline.

Non innocent full abstraction. Our goal is to show that the well-bracketed (but not necessarily innocent) model enjoys the definability property as well as (a variant of) the definable separability property with respect to IA, and hence is also fully abstract (modulo a very reasonable quotient).

The method used for proving definability is a *factorization* method. Every well-bracketed strategy σ of type A can be written as $\tau\ \texttt{cell}$, where τ is a well-bracketed *and innocent* strategy of type $\texttt{var} \to A$. The definability follows, exploiting the definability property relative to PCF, adapted to the present language (without the $\texttt{new}$ construct): there exists a term M of this sublanguage such that $\tau = [\![M]\!]$. If $A = \texttt{comm}$, we have $\sigma = [\![\texttt{new}\, x.Mx]\!]$ (for other types, see Exercise 6.7).

The factorization tells us that the essence of non-innocence is concentrated on the strategy $\texttt{cell}$. The factorization method has been used to prove the definabillity property for other languages (and other variations of game models):

- Relaxing well-bracketing but not innocence, one obtains (Laird 97 [71]) the definability property with respect to a language called μPCF (a language closely related to PCF+`catch`, cf section 3). Unfortunately, this game model does not satisfy definable separability, hence it is not fully abstract. It remains that the factorization property is interesting for itself and tells us that the essence of non well-bracketing is concentrated on the strategy `catch` (see for example [5]).

- Relaxing determinism leads to yet another factorization result: all the nondeterminism can be concentrated on a strategy called "dice" [54, 36].

The factorization that interests us here is based on a one-to-one codification of the plays s of A as (natural) numbers $code(s)$. From σ, we construct the following set τ of even-length plays on $\mathtt{var} \to A$, generated by the following rule:

$$\frac{t \in \tau \quad sab \in \sigma}{t\, a\, \mathtt{read}\, code(s)\, \mathtt{write}(code(sab))\, \mathtt{OK}\, b \ \in\ \tau}$$

(with of course also $t\ a\ \mathtt{read} \in \tau$ and $t\ a\ \mathtt{read}\ code(s)\ \mathtt{write}(code(sab)) \in \tau$). To see that this yields a (well-bracketed) deterministic and innocent strategy, note that the nondeterminism in the rule is only on the O moves a and $code(s)$ (which belong to the P view $\ulcorner ta \urcorner\, \mathtt{read}\, code(s)$ of $t\, a\, \mathtt{read}\, code(s)$), while the P move $\mathtt{write}(code(sab))$ is entirely determined by the previous O moves a and $code(s)$ and by the determinism of σ.

We are only left with checking that $\tau\ \mathtt{cell} = \sigma$. The interaction consists in the development of an even play t of τ such that the restriction $t \restriction_{\mathtt{var}}$ of t to `var` belongs to `cell`. The idea is that while s and t have a priori no relation one with the other in the assumption of the above rule, the cell discipline will force t to follow σ, much like the cell discipline enforced non interference in the example above.

We show the following properties *together*:

1. $t \restriction_A \in \sigma$, and

2. the last value written in t is (the codification of) $t \restriction_A$,

by induction on $t = t'\ a\ \mathtt{read}\ code(s)\ \mathtt{write}(code(sab))\ \mathtt{OK}\, b$. We have $t \restriction_A = t' \restriction_A\ a\, b$. The inductive hypothesis (2) applied to t' and the cell discipline force $s = t' \restriction_A$, and the hypothesis $sab \in \sigma$ of the rule reads as $t \restriction_A \in \sigma$, hence we have proved (1) for t. Moreover, (the codification of) $t \restriction_A$ is the last value written in t, i.e., property (2) holds for t. Hence we have proved that $(\tau\ \mathtt{cell}) \subseteq \sigma$.

Conversely, if $s \in \sigma$, it is possible to follow $s = s'ab$, inserting

$$\mathtt{read}\, code(s')\ \mathtt{write}(code(s'ab))\ \mathtt{OK}$$

between a and b: we obtain $t \in \tau$ such that $t \restriction_A = s$, and hence we have $\sigma \subseteq (\tau\ \mathtt{cell})$.

Now that we have the definability property, let us see if we can have the separation property. The answer is: "quasi". We shall call *complete* a play in which all the questions have been answered. A typical play that is not complete is $q_\epsilon\, q_1$. Let $comp(\sigma)$ be the set of the complete plays of a strategy σ. We shall prove the following separation property.

Proposition 6.1 *For all strategies σ_1, σ_2 of the same type A, the following equivalence holds:*

$$comp(\sigma_1) \neq comp(\sigma_2) \quad \Leftrightarrow \quad \exists\, \tau : A \to \mathtt{nat}\ \ (\tau\sigma_1 \neq \tau\sigma_2)$$

PROOF. Let, say, $m_1 \ldots m_{2n} \in comp(\sigma_1) \setminus comp(\sigma_2)$. We define τ as the strategy that contains only

$$q\, m_1 \ldots m_{2n}\, 0$$

and its even prefixes. A strategy defined in this way is not innocent in general (even if $m_1 \ldots m_{2n}$ was a view, $q\, m_1 \ldots m_{2n}\, 0$ would not be a view in general). But τ is well-bracketed (thanks to the hypothesis that $m_1 \ldots m_{2n}$ is a complete play). It is clear that $\tau\sigma_1 = 0$ and $\tau\sigma_2 = \perp$ (for the second equality, note that $\tau\sigma$ ends with an answer to q if and only if $m_1 \ldots m_{2n} \in \sigma$).

Conversely, if $\tau\sigma_1 \neq \tau\sigma_2$, then we have, say, $\tau\sigma_1 = 0$ and $\tau\sigma_2 \neq 0$. The interaction underlying $\tau\sigma_1$ is of the form $q\, m_1 \ldots m_{2n}\, 0$, where $m_1 \ldots m_{2n}$ belongs to $comp(\sigma_1)$ and cannot belong to $comp(\sigma_2)$ because this would contradict $\tau\sigma_2 \neq 0$. $\Box$

This separation property, together with the definability property and computational adequacy (we refer to [6] for the proof of this property) implies the following characterization of the observational equivalence:

$$\forall\, M, N : A\ \ (comp([\![M]\!]) = comp([\![N]\!]))\ \Leftrightarrow\ (M =_{obs} N)$$

with respect to IA and to the model of well-bracketed strategies (see Exercise 6.8).

All of this is nice, but we have thrown a small "uglyness" under the carpet. As the strategies are not innocent anymore, it is not true anymore that the only strategies of nat are the natural numbers plus $\perp$. For example, as things stand, $q\ 0\ q\ 0$ and its prefixes form a strategy, different from the strategy formed from $q\,0$ and its prefixes (observe that neither of these strategies is innocent, because the innocence forces to have $q0q0$, but also $q0q0q0$, etc..., as soon as there is $q0$). But we don't want to have other strategies than $\perp$ and n ($n \in \omega$) in type nat. For guaranteeing this, we introduce an additional component in the definition of arena. In addition to the enabling relation, the specification of an arena contains now a (prefix-closed) set P_A of authorized plays, called *valid* plays. For the arena nat, we set

$$P_{\mathtt{nat}} = \{\epsilon\} \cup \{q\} \cup \{q\, n \mid n \in \omega\}$$

For comm, we set likewise $P_{\mathtt{comm}} = \{\epsilon, \mathtt{run}, \mathtt{run\ done}\}$. But for the arena var, we place all the possible plays in $P_{\mathtt{var}}$. We refer to Exercise 6.11 for the definition of $P_{A \to B}$. The component P_A is used to limit the definition of strategy. A strategy σ is as before, and in addition it must consist only of valid plays, i.e., $\sigma \subseteq P_A$. It should be clear now that we have restored the fact that the strategies of `nat` are $\perp$ and n ($n \in \omega$).

Exercise 6.1 *Enumerate and justify all the possible values of x after the execution of* $((x := 0); (x := x + 1)) \| (x := 2)$.

Exercise 6.2 *Verify in detail that* $[\![x ::=!y + 2]\!] = (::=) \circ (id \times ((\_ + 2) \circ !))$.

Exercise 6.3 *Describe step by step the interaction of* $((x := 0); (x := x + 1))$ *and* `cell`.

Exercise 6.4 *The proof of factorization given above does not take initialization into account: as written, the definition of τ is not conform to the definition of* `cell` *that we have given. Modify either the definition of τ or the definition of* `cell` *in order to remedy to this discrepancy.*

Exercise 6.5 *Show in detail that τ is deterministic, innocent, and well-bracketed.*

Exercise 6.6 *Describe a strategy of type* $(\mathtt{nat}_{11} \to \mathtt{comm}_1) \to \mathtt{nat}_2 \to \mathtt{var}_\epsilon$ *that formalizes the idea that an imperative variable is an abstract data type with reading and writing capabilities.*

Exercise 6.7 *Given a strategy σ of type A, discuss how in general one can deduce from* $\sigma = [\![M]\!]$ `cell` *the existence of a term N of the language IA such that $\sigma = [\![N]\!]$. (In the case $A =$* `var`*, use a constant* `mkvar` : (`nat` $\to$ `comm`) $\to$ `nat` $\to$ `var` *interpreted as the strategy of Exercise 6.6).*

Exercise 6.8 *Prove the equivalence* $(comp([\![M]\!]) = comp([\![N]\!])) \Leftrightarrow (M =_{obs} N)$ *(adapting the proof of Proposition 3.2).*

Exercise 6.9 *Prove that the PCF terms M_1 and M_2 corresponding to lror and rlor are observationally equivalent in PCF, but* are not *observationally equivalent when viewed as terms of IA (find an IA context separating them).*

Exercise 6.10 *Show, as in Exercise 6.9, that the non stuttering programs of Exercise 2 are observably equivalent in PCF, and not observationally equivalent in IA, to the respective stuttering ones.*

Exercise 6.11 *We want to prove that the addition of the component P_A is "conservative" over the innocent model of PCF, in the sense that for what regards the interpretation of PCF, this component is superfluous. More acurately, the interpretation of a term M of PCF is the same in the model without the component P and in the model with the component P.*

1. *Justify the fact that for this to hold it is enough to prove that at any PCF type, all views belong to P_A.*

2. *Prove that this latter property holds. The definition of $P_{A \to B}$ is the following: $s \in P_{A \to B}$ iff $s \restriction_B \in P_B$ and $s \restriction_m \in P_A$ for all initial move m of A, where $s \restriction_m$ is the subsequence of s that contains all the moves of s that are hereditarily justified by m (in particular, $s \restriction_m$ consists only of moves of A). (This definition is easier to grasp if one thinks of $A \to B$ as $(!A) \multimap B$ in the sense of linear logic.)*

Exercise 6.12 *Enumerate all the strategies of* `comm` $\to$ `comm`.

1. *Prove that there exists a bijective correspondence between these strategies and the "lazy" natural numbers , given by the following syntax:*

$$l ::= \Omega \mid 0 \mid S(l)$$

to which one adds the unique infinite term $S(S(S(\ldots$ that can be obtained from this syntax. Describe the partial order structure on these "numbers" inherited from the inclusion of strategies. (In order to get some intuition on "lazy" numbers, think of $S(S(0))$ as 2, and a $S(S(\bot))$ as a number that is not yet defined but that will be ≥ 2 when it will be defined.)

2. *For every strategy σ of* `comm` $\to$ `comm`, *give a term M of IA (possibly using Ω and Y) that defines it.*

3. *Prove that for these terms we have $M =_{obs} N$ iff $[\![M]\!] = [\![N]\!]$ or*

$$\exists\, m, n \;\; [\![M]\!] = S^m(\bot) \;\text{ and }\; [\![N]\!] = S^n(\bot)$$

(where $m, n \in \omega \cup \{\infty\}$, $S^0(\bot) = \bot$, $S^{m+1}(\bot) = S(S^m(\bot))$, and $S^\infty(\bot) = S(S(S(\ldots)$.

7 Applications to abstract interpretation

This section is based on [42, 37]. Ghica and McCusker made the following observations. When the types are at most second-order, pointers are useless, i.e., they can be reconstructed uniquely. Morevover, the strategies (more precisely, their underlying sets of complete plays) interpreting IA terms, considered as sets of words (a play without pointers is just a word of moves) are regular languages. It follows that observational equivalence is decidable for this second-order fragment of IA, since the equality of regular languages (or of regular expressions) is decidable.

We show here how to (re-)express the meaning of some programs in terms of regular languages. An auxiliary notation will be useful. For every term M of type `comm`, $comp([\![M]\!])$ is a set of words that all begin with `run` and end with `done`. Therefore, it is natural to write

$$comp([\![M]\!]) = \mathtt{run} \cdot (\!|M|\!) \cdot \mathtt{done}$$

The simplest example is the command `skip` (whose execution does not change the state): $(\!|\mathtt{skip}|\!) = \epsilon$. Sequencing becomes just ... sequencing:

$$(\!|M; M'|\!) = (\!|M|\!) \cdot (\!|M'|\!)$$

In the same spirit, for a term of type `nat` (or `bool`), we write $comp([\![M]\!]) = \Sigma_{n\in\omega} q \cdot (\!|M|\!)_n \cdot n$ with

$$(\!|n|\!)_m = \begin{cases} \epsilon \text{ if } m = n \\ \emptyset \text{ if } m \neq n \end{cases}$$

The semantics of the classical `while` and `if then else` command constructs are quite elegant in this setting:

$$\begin{array}{l} (\!|\mathtt{while}\, M \,\mathtt{do}\, N|\!) = ((\!|M|\!)_T \cdot (\!|N|\!))^* \cdot (\!|M|\!)_F \\ (\!|\mathtt{if}\, M \,\mathtt{then}\, N \,\mathtt{else}\, P|\!) = (\!|M|\!)_T \cdot (\!|N|\!) + (\!|M|\!)_F \cdot (\!|P|\!) \end{array}$$

(recall that in regular expressions $+$ denotes union, and if L is a language, then $L^* = L + L \cdot L + L \cdot L \cdot L + \ldots$). The * construction is also useful to describe copycat strategies (in “fat” form, cf. section 2):

$$\begin{array}{l} [\![f : \mathtt{comm} \to \mathtt{comm} \vdash f : \mathtt{comm} \to \mathtt{comm}]\!] = \\ \quad \mathtt{run}_\epsilon \cdot \mathtt{run}_2 \cdot (\mathtt{run}_{21} \cdot \mathtt{run}_1 \cdot \mathtt{done}_1 \cdot \mathtt{done}_{21})^* \cdot \mathtt{done}_2 \cdot \mathtt{done}_\epsilon \end{array}$$

In this setting, application becomes substitution of languages for strings. For example, we have

$$comp([\![\lambda x.x + 1]\!]) = \Sigma_{n\in\omega} q_\epsilon q_1 n_1 (n+1)_\epsilon$$

and for every M of type `nat`,

$$comp(([\![\lambda x.x + 1]\!])M) = (\Sigma_{n\in\omega} q_\epsilon q_1 n_1 (n+1)_\epsilon)[\ldots, (q_1 n_1) \leftarrow (\!|M|\!)_n, \ldots]$$

We leave the reader check that for, say $M = 7$, this computation gives indeed $q_\epsilon 8_\epsilon$. Here is the regular expression for `cell`:

$$(\Sigma_{n\in\omega} \mathtt{write}(n) \cdot \mathtt{OK} \cdot (\mathtt{read} \cdot n)^*))^*$$

In the exercises, we list some examples of observationally equivalent programs. In each case, the exercise amounts to compute (the set of complete plays of) the interpretation of each of the two terms and to show that they are equal.

We now prove that we can dispense with pointers. The proof relies on some invariants that we did not consider so far, because they can be left implicit in the meager (view function) setting.

Visibility. It can be shown by induction on IA types that all the plays in P_A (for all arenas A interpreting an IA type) respect the following so-called *visibility* conditions:

- For all even-length $sm \in P_A$, m points to a move that belongs to $\ulcorner s \urcorner$.
- A dual condition holds for all odd-length plays $sn \in P_A$, namely, n points to a move that belongs to the *O view* of s (whose definition is quite symmetric to that of P view).

It is easy to check that P views satisfy these conditions vacuously.

Well-opened games. It can also be shown by induction on IA types that all the plays in P_A contain at most one occurrence of each initial move. A game with this property is called *well-opened* [85]. Again, views always satisfy this property.

Proposition 7.1 *Let* $A = A_1 \to \ldots \to A_n \to \texttt{base}$*, where* $A_i = \texttt{base}_{i1} \to \ldots \to \texttt{base}_{il_i} \to \texttt{base}_i$ *for all i, be a second-order IA type, and let* σ *be a well-bracketed strategy. Then all the pointers of all the plays of* σ *are uniquely determined.*

Proof. By well-bracketing, all the answer moves must point to the closest occurrence of their justifying move. Therefore we concentrate on the question moves:

1. q_ϵ is an initial move of A and hence has no pointer.
2. q_i points to q_ϵ. Then the pointer of q_i is uniquely determined, since q_ϵ occurs only once in the play by well-openedness.
3. q_{ij} points to q_i. Since q_{ij} must point in its O view by visibility, it is enough to examine the O view of a play sq_{ij} of σ. In an O view, q_i must occur immediately after q, and therefore occurs uniquely. It follows that the pointer from q_{ij} is uniquely determined. $\square$

There is still a problem with using the regular language semantics to prove program properties effectively: the decidability of equivalence of regular expressions holds only for *finite* alphabets. But the arena `nat` has an infinite number of moves. Whence the idea of using abstract interpretation, and to replace in typing judgments all occurrences of the type `nat` (and of the type `var`) by finite approximations, while the syntax of IA terms remains the same.

A finite approximation of `nat` or `var` (notation $\texttt{nat}_\pi$, $\texttt{var}_\pi$) is specified by a partition π of ω into a finite number of equivalence classes. For example, $[0, 5]$ denotes the partition

$$\{0\}\ ,\ \{1\}\ ,\ \ldots\ ,\ \{5\}\ , \{n \mid n > 5\}$$

and [] denotes the rawest partition consisting of one class only. We say that $\mathtt{nat}_{\pi'}$ refines $\mathtt{nat}_{\pi}$ (or that $\mathtt{nat}_{\pi}$ abstracts $\mathtt{nat}_{\pi'}$) when each π' equivalence class is included in some π equivalence class. The arenas for $\mathtt{nat}_{\pi}$, $\mathtt{var}_{\pi}$ are the same as the arenas for nat, var, except that the moves n, $\mathtt{write}(n)$ are now $[n]_{\pi}$, $\mathtt{write}([n]_{\pi})$ (where $[n]$ is the equivalence class of n with respect to π). What about operations on numbers or reading and writing instructions? Each occurrence of them is interpreted according to its type. For example, when $:=$ receives the type $\mathtt{var}_{\pi'} \times \mathtt{nat}_{\pi} \to \mathtt{comm}$, its interpretation contains all the plays

$$\mathtt{run}\ q\ [n]_{\pi}\ \mathtt{write}([n]_{\pi'})\ \mathtt{OK}\ \mathtt{done}$$

Note that this is a *nondeterministic strategy* in general! For example, if $\pi = []$ and $\pi' = [0,2]$, the effect of $x := 1$ is to "write" $\{0\}$, $\{1\}$, $\{2\}$, or $\{n \mid n > 2\}$, that is, any of the possible results!

As another example of the nondeterminism arising from abstract interpretation, suppose that $=$ receives the type $\mathtt{nat}_{\pi_1} \times \mathtt{nat}_{\pi_2} \to \mathtt{bool}$. Then its interpretation contains all the plays $qq_1[n]_{\pi_1}q_2[n]_{\pi_2}T$ and all the plays $qq_1[n]_{\pi_1}q_2[m]_{\pi_2}F$ (for $m \neq n$). Then, say, as soon as the equivalence class of n is not a singleton in either π_1 or π_2, then $n = n$ may execute nondeterministically to T or F. To be completely specific, suppose that $[2]_{\pi_1} = \{2\}$ and $[2]_{\pi_2} = \{0,2\}$, then $=$ contains both $qq_1[2]_{\pi_1}q_2[2]_{\pi_2}T$ and $qq_1[2]_{\pi_1}q_2[2]_{\pi_2}F$. We say that the interaction underlying the term $2 = 2$ is nondeterministic for this choice of abstractions. It suffices to refine π_2 to π_2' so as to have $[2]_{\pi_2'} = \{2\}$ in order to obtain that $2 = 2$ is T, deterministically.

For an illustration of the use of abstract types for proving program properties, we shall turn from proofs of observational equivalence to proofs of a safety property. More specifically, we consider an extension of IA with an abort instruction (and a corresponding extension of the arena comm with a new P move abort), and we want to check whether a program is safe. Here, safe means that no execution will ever lead to an execution of abort. For example, the program

$$\mathtt{new}\ v.\ v := 0; f(v :=!v + 1, \mathtt{if}\ !v = 5\ \mathtt{then}\ \mathtt{abort}\ \mathtt{else}\ \mathtt{skip})$$

is unsafe, because f might call its first argument five times and then call its second argument. But the program

$$\mathtt{new}\ v.\ v := 3\ ;\ \mathtt{if}\ !v =!v + 1; \mathtt{then}\ \mathtt{abort}\ \mathtt{else}\ \mathtt{skip}$$

is safe, provided we choose an abstract interpretation which is not too raw (it has to be refined enough for 3 and 4 not to belong to the same equivalence class).

These two situations (and the above example of refinement for $=$) suggest the following loop for checking a second-order IA term M for safety.

1. Evaluate the meaning of M with respect to some abstract interpretation (obtained by decorating all the occurrences of the types `nat` and `var` in the typing derivation of M).

2. If this abstract meaning of M does not contain the move `abort`, conclude that M is safe.

3. If it contains an occurrence of `abort`, then

   (a) if the interaction that has led to this `abort` is deterministic, then conclude that M is unsafe;

   (b) if all the occurrences of `abort` have been reached nondeterministically, then refine the abstract interpretation accordingly, and go back to step 1.

A refined abstract interpretation is obtained by replacing all the decorations in the typing judgment by refined abstractions. A precise version of this algorithm has been proposed and implemented by Ghica and his coauthors. They have shown that it gives a semi-decision procedure: if the program is unsafe, the algorithm will discover it [37] (see Exercise 7.8). The correctness of the algorithm relies on three properties:

1. A correspondence between operational and denotational semantics: a term may provoke an abort in some context if and only if its denotational semantics in the (non abstracted) game model contains a play ending with `abort`.

2. It two decorations for a typing judgment are given, one refining the other, and if a play appears in the more refined interpretation, then its abstraction (i.e., the same play in which all equivalence classes are abstracted according to the less refined decoration) appears in the more abstract interpretation. Hence, once an abstract meaning is recognized to be safe, we know that the term is safe.

3. It two decorations for a typing judgment are given, one refining the other, and if a play appears in the less refined interpretation of a term and if the interaction that has led to this play is deterministic, then this play lifts to (i.e., is an abstraction of) a play of the more refined interpretation. Hence, once an abstract meaning contains an `abort` that has been obtained deterministically, then we know that the term is unsafe.

Exercise 7.1 *Show that* `while` T `do` $M =_{obs}$ `diverge`*, where* `diverge` *is a self-explanatory command.*

Exercise 7.2 *Show that* `new` $v.c \ =_{obs}\ c$ *in the context* $c :$ `comm` *(locality).*

Exercise 7.3 *Prove that in the context* $f : \texttt{comm} \to \texttt{comm}$

$$\texttt{new}\, v.(v := 0\ ;\ f(v := 1)\ ;\ (\texttt{if}\ !v = 1\ \texttt{then diverge else skip}))$$

(cf. Exercise 7.1) is observationally equivalent to $f(\texttt{diverge})$ *(irreversibility of state change).*

Exercise 7.4 *Prove that in the context* $f : \texttt{comm} \to \texttt{comm}$

$$\texttt{new}\, v.(v := 0\ ;\ f(v :=!v + 2)\ ;\ (\texttt{if}\ !v\ \texttt{mod}\ 2 = 0\ \texttt{then diverge else skip}))$$

is observationally equivalent to `diverge` *(invariant preservation).*

Exercise 7.5 *Prove that in the context* $f : \texttt{comm} \to \texttt{bool} \to \texttt{comm}$

$$\texttt{new}\, v.(v := 0; f(v := 1)(!v = 0))\ =_{obs}\ \texttt{new}\, v.(v := 0; f(v := -1)(!v = 0))$$

(two possible implementations of a switch object).

Exercise 7.6 *Prove that in the context* $f : \texttt{comm} \to \texttt{comm}$

$$(\texttt{new}\, v.f(v :=!v + 1))\ =_{obs}\ f(\texttt{skip})$$

(parametricity).

Exercise 7.7 *Check on all examples of executions given in section 2 that the interactions involved respect the visibility and well-openedness conditions.*

Exercise 7.8 *Consider the term*

$$y : \texttt{nat} \vdash \texttt{new}\, v.\ v := y\ ;\ \texttt{if}\ !v =!v + 1; \texttt{then abort else skip}$$

and give evidence of the fact that Ghica and al's algorithm provides only a semi-decision procedure.

8 Proofs as strategies.

The interactive interpretation provided by game semantics applies as well to the semantics of proofs, in various settings: intuitionistic, classical, and in various fragments or variants of linear logic. As usual, we have no pretention of exhaustivity here. Some of these interpretations have already been cited in section 1.

Dialogue games. We recall here that an entry point for the early work of Lorenzen and his coworkers is [41]. It should be pointed out that the distinction (and equivalence) between the "meager" and "fat" points of view (cf. section 2), i.e., between the view functions and the strategies (cf. section 4), can be traced back to the distinction made in [41] between E-dialogues and D-dialogues (in Felscher's words, in an E-dialogue, O "can react only upon the immediately preceding" P move).

Classical logic. The use of dialogue games was deeply renewed when Coquand [24] presented a version of classical logic (based on infinitary conjunctions and disjunctions to interpret quantifications over numbers) where cut-elimination matches step by step the interaction (expressed in the framework of the View Abstract Machine [24]) of the proofs viewed as strategies. This correspondence was further investigated by Herbelin [55, 56]. Further, it can be composed with the Curry-Howard isomorphism, so that we get close correspondences between the three following families of devices [35, 29, 32, 30]:

- cut-elimination procedures,
- devices to describe the interaction of strategies, such as the View Abstract Machine or the Geometric Abstract Machine (cf. section 2),
- abstract machines for the execution of programming languages, such as Krivine abstract machine.

Linear logic. As we mentioned in the introduction (and in section 3), the arise of game semantics for PCF was immediately preceded by full completeness results for multiplicative linear logic [3, 60]. Since then, several works have proposed models for larger fragments of linear logic [11, 8, 87].

The interpretation of negation is very intuitive: the dual $A^{\perp}$ of A is A in which one has just exchanged the roles of Player and Opponent, keeping the underlying set of moves and the enabling relation unchanged. Notice that this seemingly innocuous construction breaks the invariant respected so far, namely that it is always Opponent who starts. So, there are now arenas where Opponent starts, and arenas where Player starts (or even possibly arenas where some plays may start with Opponent, and other plays may start with Player). Composition of strategies in this extended setting becomes tricky, and is not associative in general (in a deterministic setting). This is known as the Blass problem (see e.g. [8]). In [11], the problem is circumvented by working with nondeterministic strategies. In [8], the problem is circumvented by a definition of strategy inspired from the idea of playing several moves at once. In [76, 78], Laurent shows give a fully complete semantics for a *polarized* version of linear logic, based on the division of connectives of linear logic into negative ones ($⅋$,$\&$) and positive ones ($\otimes$,$\oplus$), that goes back to Andreoli [10]. The "of course" (resp. "why not") connective must be applied to a negative (resp.

positive) formula and yields a positive (resp. negative) formula. Positive formulas are interpreted by positive arenas, which are arenas in which all initial moves are P moves, while negative formulas are interpreted dually by negative arenas, which are the arenas that we have considered so far. In this restricted setting, composition behaves well, with (a straightforward adaptation of) the (deterministic) HO framework. Here, we shall only discuss Laurent's interpretation of "of course" and $⅋$ (see Exercise 8.1).

- The meaning of $!A$ has two aspects: it allows for multiple uses of resources of type A, and it also performs a change of polarity. This leads to the following decomposition:

  $$!A = \downarrow \# A$$

  with the following interpretation:

  – Given an arena A with last component P_A (cf. section 6), $\# A$ is the arena that has the same underlying set of moves, with the same polarity, and the same enabling relation, and whose last component is defined as follows:

  $$P_{\# A} = \{s \mid s \restriction_m \in P_A, \text{ for all initial moves } m \text{ of } A\}$$

  where $s \restriction_m$ is the subsequence of s that contains all the moves of s that are hereditarily justified by m (in particular, $s \restriction_m$ consists only of moves of A) (cf. Exercise 6.11). Note that A and $\# A$ have the same polarity.

  – $\downarrow A$, where A is negative, is obtained by adding one P move $*$ to A, with $\vdash *$, $* \vdash m$ for all initial moves of A, and the rest of the enablings as in A. The component P_A is obtained by prefixing $*$ to all the plays of P_A. The resulting arena is positive.

  (In [85], $!A$ is defined as $\# A$, and the construction works – i.e., satisfies the axiom of a categorical model of linear logic (see e.g. [88]) – only for well-opened games (cf. section 7). Unfortunately, if A is well-opened, $\# A$ is not well-opened (by construction), hence negative games do not quite provide a model of (intuitionistic) linear logic. Laurent's framework repairs this and provides a categorical model of *polarized* linear logic (and even of a slightly larger logical system called LLP), i.e., a control category [76]).

- $A ⅋ B$ is the arena that has as initial O moves *pairs* (m_1, n_2) where m is initial in A and n is initial in B, and otherwise has as moves all the other moves of A and B, i.e., it has all m_1 such that m is a non initial move of A and all n_2 such that n is a non initial move of B. The enabling relation is as follows:

  $$\frac{\vdash_A m \quad \vdash_B n}{\vdash (m_1, n_2)} \quad \frac{\vdash_A m \quad m \vdash_A a}{(m,n) \vdash a_1} \quad \frac{\vdash_B n \quad n \vdash_B b}{(m,n) \vdash b_2} \quad \frac{\not\vdash_A a \quad a \vdash_A a'}{a_1 \vdash a'_1} \quad \frac{\not\vdash_B b \quad b \vdash_B b'}{b_2 \vdash b'_2}$$

Two kinds of "of course". As we have just suggested, the underlying linear decomposition of the function space construction in the HO game semantics relies on a definition of ! (actually, of #) which interprets multiple use of resources by the interleaving of arbitrarily many threads of computation. In particular, while `bool` has finitely many strategies, this is not true anymore of `bool` $\to$ `bool`, since it contains, not only $\lambda x.\texttt{case}\ x\ [T \to T]$, but also, say, $\lambda x.\texttt{case}\ x\ [T \to \texttt{case}\ x\ [T \to T]]$, $\lambda x.\texttt{case}\ x\ [T \to \texttt{case}\ x\ [T \to \texttt{case}\ x\ [T \to T]]]$, etc...

On the other hand, the model of sequential algorithms relies on a different linear decomposition, which induces a quotient on such "stuttering" terms (cf. section 2). The underlying paradigm is *memorization* rather than *repetition*: the idea is that strategies memorize which moves they do. There are only finitely sequential algorithms from `bool` to `bool` (see Exercise 8.2). As a matter of fact, the morphisms of the model of sequential algorithms can be described as functions over domains extended with error elements [20, 74] (see Exercice 8.3).

This difference between HO games and sequential algorithms is reminiscent of two of the possible choices for the "of course" in coherence space semantics: the multiset-based one and the set-base one, respectively. Recall that a coherence space is given by a set of tokens E together with a coherence relation. (Each arena gives rise to a coherence space, whose tokens are the plays, and where two plays are coherent when their longest common prefix is of even-length.) In [45], the tokens of $!E$ are taken to be the finite cliques of E (i.e. the finite sets of tokens which are pairwise coherent). In more recent works (see e.g. [48]), the tokens of $!E$ are often taken to be finite *multisets* of tokens. The first (resp. the second) definition yields a finite (resp. infinite) $!E$ from a finite E (where finite means having finitely many cliques).

Ludics. Ludics [52] proposes a revisiting of logical principles in which interaction is built-in from the beginning. The formalism involves strategies (called *designs*) of an untyped, universal arena (see [39] for a "dictionary" between ludics and game semantics).

Quite like in the old days of semantics when untyped λ-calculus was the core object of study and when types were interpreted as suitable sets of λ-terms (see e.g. [57]), in ludics, formulas are interpreted as well-behaved sets of designs, called *behaviours*. A full completeness result of this semantics with respect to a polarized version of multiplicative-additive linear logic, is proved in [52].

Exercise 8.1 *Check that the definitions above of #, $\downarrow$, and $⅋$ agree with the definition of function space that we have used in section 2:*

$$A \to B = (!A) \multimap B = (!A)^{\perp} ⅋ B = (\downarrow \# A)^{\perp} ⅋ B$$

where the first equalities are the well-known linear decomposition of the function space at the origin of linear logic [45].

Exercise 8.2 *List all the sequential algorithms of* `bool` $\rightarrow$ `bool` *and of* `bool` $\times$ `bool` $\rightarrow$ `bool` *(written using the language of HO arenas). Which of these algorithms make multiple use of their resources?*

Exercise 8.3 *Show that all sequential algorithms of Exercise 8.2 can be mapped injectively to functions from* $\mathbf{B}_{err}$ *to* $\mathbf{B}_{err}$ *and from* $\mathbf{B}_{err} \times \mathbf{B}_{err}$ *to* $\mathbf{B}_{err}$*, respectively, where* $\mathbf{B}_{err} = \{\perp, T, F, err\}$.

References

[1] Abadi, M., L. Cardelli, P.-L. Curien, and J.-J. Lévy, *Explicit substitutions*, Journal of Functional Programming **1**(4) (1992), 375-416.

[2] Abramsky, S., *Domain theory in logical form*, Annals of Pure and Applied Logic **51** (1991), 1-77.

[3] Abramsky, S., and R. Jagadeesan, *Games and full completeness for multiplicative linear logic*, Journal of Symbolic Logic, **59**(2) (1994), 543–574 (conference version in Proc. FOSSACS'92).

[4] S. Abramsky, R. Jagadeesan and P. Malacaria, *Games and full abstraction for PCF*, Information and Computation 163 (2), 409-470 (2000) (preliminary conference version in Proc. Theoretical Aspects of Computer Science 1994, Springer LNCS 789, 1-16).

[5] Abramsky, S., and G. McCusker, *Game Semantics*, in H. Schwichtenberg and U. Berger, editors, Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School, Springer-Verlag (1998).

[6] Abramsky, S., and G. McCusker, *Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions*, in O'Hearn and Tennent (eds.), Algol-like languages, volume 2, Progress in Theoretical Computer Science, 297-329 (1997), Birkhäuser.

[7] Abramsky, S., K. Honda, and G. McCusker, *A fully abstract game semantics for general references*, in Proc. 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99), IEEE Computer Society Press, 431-442 (1999).

[8] Abramsky, S., and P.-A. Melliès, *Concurrent games and full completeness*, Proc. 14th annual IEEE Symposium on Logic in Computer Science (LICS'99), IEEE Computer Society Press, 431-444 (1999).

[9] Amadio, R., and P.-L. Curien, "Domains and Lambda-Calculi", Cambridge University Press (1998).

[10] Andreoli., J.-M., *Focussing and proof construction* Annals of Pure and Applied Logic **107**(1) (2001), 131-163.

[11] Baillot, P., V. Danos, T. Ehrhard, and L. Regnier, *Believe it or not, AJM's games is a model of classical linear logic*, Proc. 12th annual IEEE Symposium on Logic in Computer Science (LICS'97), IEEE Computer Society Press, 68-75 (1997).

[12] Berry, G., *Stable models of typed lambda-calculi*, Proc. International Colloquium on Automata, Languages and Programming (ICALP'78), Springer Lect. Notes in Comp. Sci. 62 (1978).

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

[14] Berry, G., and P.-L. Curien, *The kernel of the applicative language CDS: theory and practice*, Proc. French-US Seminar on the Applications of Algebra to Language Definition and Compilation, Cambridge University Press (1985).

[15] Blass, A., *Degrees of indeterminacy of games*, Fundamenta Mathematica **LXXVII** (1972), 151-162.

[16] Blass, A., *A game semantics for linear logic*, Annals of Pure and Applied Logic **56** (1992), 183-220.

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

[18] Bucciarelli, A., *Another approach to sequentiality: Kleene's unimonotone functions*, in Proc. Mathematical Foundations of Programming Semantics (MFPS'93), Springer Lecture Notes in Computer Science 802 (1993).

[19] Cartwright R., and M. Felleisen, *Observable sequentiality and full abstraction*, Proc. ACM Principles of Programming Languages 1992.

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

[21] Chroboczek, J. , *Game semantics and subtyping*, in Proc. of the 15th annual IEEE Symposium on Logic in Computer Science (LICS'00).

[22] Conway, J.H., "On numbers and games", Academic Press (1976), Vol. 6 of London Mathematical Society Monograph.

[23] Coppo, M., and M. Dezani, *A new type assignment for lambda-terms*, Archiv. Math. Logik **19** (1978), 139-156.

[24] Coquand, T., *A semantics of evidence for classical arithmetic*, Journal of Symbolic Logic 60, 325–337 (1995).

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

[26] Curien, P.-L., "Combinateurs catégoriques, algorithmes séquentiels et programmation fonctionnelle", Thèse d'Etat, Univ. Paris 7 (1983).

[27] Curien, P.-L., "Categorical Combinators, Sequential Algorithms and Functional Programming", Pitman (1986), revised edition, Birkhäuser (1993).

[28] Curien, P.-L., *On the symmetry of sequentiality*, Proc. Mathematical Foundations of Programming Semantics (MFPS'93), Springer Lect. Notes in Comp. Sci. 802, 29-71 (1994).

[29] Curien, P.-L., *Abstract Böhm trees*, Mathematical Structures in Computer Science **8**(6) (1998), 559-591.

[30] Curien, P.-L., and H. Herbelin, *Computing with abstract Böhm trees*, Third Fuji International Symposium on Functional and Logic Programming, April 1998, Kyoto, World Scientific (Singapour) (1998), 20-39.

[31] Curien, P.-L., *Sur l'η-expansion infinie*, Comptes-Rendus de l'Académie des Sciences **334** Sec. I (2002), 77-82.

[32] Curien, P.-L., and H. Herbelin, *Abstract machines for dialogue games*, in "Interactive models of computation and program behaviour", Panoramas et Synthèses 27, Société Mathématique de France (2009).

[33] Curien, P.-L., and C. Faggian, *An approach to innocent strategies as graphs*, in Proc. Computer Science Logic (CSL'05) (2005).

[34] P.-L. Curien, Definability and full abstraction, in Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin, Electronic Notes in Theoretical Computer Science 172, 301-310 (2007).

[35] Danos, V., H. Herbelin, and L. Regnier, *Game semantics and abstract machines*, in Proc. 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96).

[36] Danos, V., and R. Harmer, *Probabilistic game semantics*, ACM Transactions on Computational Logic **3**(3) (2002).

[37] Dimovski, A., D. R. Ghica, and R. Lazic. *Data-abstraction refinement: A game semantic approach*, in Proceedings of the 12th International Static Analysis Symposium (SAS'05), London (2005).

[38] Faggian, C., *Travelling on designs: ludics dynamics*, Proc. Computer Science Logic (CSL'02), Lecture Notes in Computer Science 2471 (2002).

[39] Faggian, C., and M. Hyland, *Designs, disputes and strategies*, Proc. Computer Science Logic (CSL'02), Lecture Notes in Computer Science 2471 (2002).

[40] Faggian, C., and F. Maurel, *Ludics nets, a game model of concurrent interaction*, in Proc. of 20th annual IEEE Symposium on Logic in Computer Science (LICS'05), Computer Society Press (2005).

[41] W. Felscher, *Dialogues as a foundation of intuitionistic logic*, Handbook of Philosophical Logic **3**, 341-372 (1986).

[42] Ghica, D.R., and Guy McCusker, *The regular-language semantics of second-order idealized ALGOL*, Theoretical Computer Science **309**(1-3) (2003), 469-502.

[43] Ghica, D.R., and A. Murawski, *Angelic Semantics of Fine-Grained Concurrency*, in Proc. of Foundations of Software Science and Computation Structures (FOSSACS'04), Lecture notes in Computer Science 2987 (2004).

[44] Girard, J.-Y., *The system F of variable types, fifteen years later*, Theoretical Computer Science **45** (1986), 159-192.

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

[46] Girard, J.-Y., *Geometry of interaction I: interpretation of system F*, in Proc. Logic Colloquium 1988, R. Ferro et al. eds, 221-260 (1989), North Holland.

[47] Girard, J.-Y., *Geometry of interaction II: deadlock-free algorithms*, Proc. COLOG-88, P. Martin-Löf and G. Mints eds, Springer Lecture Notes in Computer Science 417 (1990), 76-93.

[48] Girard, J.-Y., *Linear Logic, its syntax and semantics*, in Advances of Linear Logic, J.-Y. Girard, Y. Lafont and L. Regnier eds, London Mathematical Society Lecture Notes Series 222 (1995), Cambridge University Press.

[49] Girard, J.-Y., *Geometry of interaction III: accommodating the additives*, in Advances of Linear Logic, J.-Y. Girard, Y. Lafont and L. Regnier eds, London Mathematical Society Lecture Notes Series 222 (1995), Cambridge University Press.

[50] Girard, J.-Y. *Geometry of interaction IV : the feedback equation*, manuscript (version of December 2005 available from `http://iml.univ-mrs.fr/~girard`).

[51] Girard, J.-Y., *On denotational completeness*, Theoretical Computer Science **227** (1999), 249-273.

[52] Girard, J.-Y., *Locus solum*, Mathematical Structures in Computer Science **11** (2001), 301–506.

[53] Harmer, R., "Games and full abstraction for nondeterministic languages", PhD Thesis, Imperial College (1999).

[54] Harmer, R., and G. McCusker, *A fully abstract game semantics for finite nondeterminism*, in Proc. of the 14th annual IEEE symposium on Logic in Computer Science (LICS'99).

[55] Herbelin, H., "Séquents qu'on calcule", Thèse de Doctorat, Université Paris VII (1995).

[56] Herbelin, H., *Games and weak-head reduction for Classical PCF*, Proc. of Typed Lambda-Calculi and Applications (TLCA'97), Lecture Notes in Computer Science 1210, 214–230.

[57] Hindley, R., *The completeness theorem for typing lambda-terms*, Theoretical Computer Science **22** (1983), 1-17.

[58] Honda, K., and N. Yoshida, *Game-theoretic analysis of call-by-value computation*, Theoretical Computer Science **221**(1-2) (1999), 393-456.

[59] Hyland, J.M.E.., *A syntactic characterization of the equality in some models of lambda calculus*, J. London Math. Soc. **2** (1976), 361-370.

[60] Hyland, J.M.E.., and C.-H. L. Ong, *Fair games and full completeness for multiplicative linear logic without the mix-rule*, preprint (1993).

[61] Hyland, J.M.E., and L. Ong, *On full abstraction for PCF I, II, and III*, Information and Computation **163**(1) (2000), 285-408 (preliminary draft circulated since 1994).

[62] Hyland, J.M.E., and A. Schalk, *Games on graphs and sequentially realizable functionals*, in Proceedings of 17th annual IEEE Symposium on Logic in Computer Science (LICS'02), Copenhagen, pages 257-264, Computer Society Press (2002).

[63] Joyal, A., *Remarques sur la théorie des jeux à deux personnes*, Gazette des Sciences Mathématiques du Québec **1**(4) (1977).

[64] Joyal, A., *Free lattices, communication and money games*, in M.L. Dalla Chiara et al. (eds), Logic and Scientific Methods, 29-68, Kluwer (1998).

[65] Kahn, G., and G. Plotkin, *Concrete domains*, Theoretical Computer Science **121** (1993), 187-277 (appeared in French as TR IRIA-Laboria 336 in 1978).

[66] Kanneganti, R,. and Cartwright, R. and Felleisen, M., *SPCF: its model, calculus, and computational power*, in Proc. REX Workshop on Semantics and Concurrency, Springer Lecture Notes in Computer Science 666 (1993).

[67] Kleene, S., *Recursive functionals and quantifiers of finite yypes revisited I*, in Proc. General Recursion Theory II, Fenstad et al. (eds.) (1978), North-Holland.

[68] Kleene, S., *Recursive functionals and quantifiers of finite types revisited II*, in Proc. Kleene Symposium, Barwise et al. (eds.) (1980), North-Holland.

[69] Kleene, S., *Recursive functionals and quantifiers of finite types revisited III*, in Proc. Patras Logic Symposium (1982), North Holland.

[70] Kleene, S., *Unimonotone functions of finite types (recursive functionals and quantifiers of finite types revisited IV)*, in Proc. Symposia in Pure Mathematics 42 (1985).

[71] Laird, J., *Full abstraction for functional languages with control*, Proc. 12th Annual Symposium on Logic in Computer Science (LICS '97) (1997).

[72] Laird, J., *A fully abstract bidomain model of Unary FPC.* in Proc. of the Int. Conf. on Typed Lambda-Calculi and Applications (TLCA'03), Lecture Notes in Computer Science 2701 (2003).

[73] Laird, J., *Games and sequential algorithms.* To appear.

[74] Laird, J., *Locally Boolean domains*, Theoretical Computer Science **342** (2005), 132-248.

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

[76] Laurent, O., *Polarized games*, Annals of Pure and Applied Logic (APAL), **130**(1-3) (2004), 79-123.

[77] Laurent, O., *Sémantique des jeux*, course notes available from `www.pps.jussieu.fr/~laurent` (2004).

[78] Laurent, O., *Syntax vs. semantics: a polarized approach*, Theoretical Computer Science **343**(1-2) (2005), 177-206.

[79] Loader, R., *Unary PCF is decidable*, Theoretical Computer Science **206**(1-2) (1998), 317-329.

[80] Loader, R., *Finitary PCF is undecidable*, Theoretical Computer Science **266**(1-2) (2001), 341-364.

[81] Longley, J., *The sequentially realizable functionals*, Annals of Pure and Applied Logic **117**(1-3) (2002), 1-93.

[82] Malacaria, P., and C. Hankin, *A new approach to control flow analysis*, in Proc. of Compiler Construction 1998, Lecture Notes in Computer Science 1383 (1998).

[83] Malacaria, P., and C. Hankin, *Generalised flowcharts and games*, in Proc. International Colloquium on Automata, Languages and Programming (ICALP'98), Lecture Notes in Computer Science 1443 (1998).

[84] Malacaria, P., and C. Hankin, *Non-deterministic games and program analysis: an application to security*, in Proc. of the 14th annual IEEE symposium on Logic in Computer Science (LICS'99).

[85] McCusker, G., "Games and full abstraction for a functional metalanguage with recursive types", PhD Thesis, Imperial College and Univ. of London (1996), published in Springer Verlag's Distinguished Dissertations in Computer Science series (1998).

[86] Melliès, P.-A. *Asynchronous games 2: the true concurrency of innocence*, in Proc. of the 15th International Conference on Concurrency Theory, London, 2004. Lecture Notes in Computer Science 3170, 448-465, Springer Verlag.

[87] Melliès, P.-A., *Asynchronous games 4: a fully complete model of propositional linear logic*, Proc. of the 21th Conference on Logic in Computer Science (LICS'05), Chicago, 2005.

[88] Melliès, P.-A., *Categorical semantics of linear logic: a survey*, to appear in the volume "Interactive models of computation and program behaviour", Panoramas et Synthèses, Société Mathématique de France.

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

[90] Nickau, H., *Hereditarily sequential functionals*, Proc. Logical Foundations of Computer Science: Logic at St. Petersburg, Eds. A. Nerode and Yu. V. Matiyasevich, Lecture Notes in Computer Science 813, 253-264 (1994).

[91] O'Hearn, P., and J. Riecke, *Kripke logical relations and PCF*, Information and Computation **120**(1) (1995), 107-116.

[92] Ong, C.-H. L., *Observational equivalence of third-order Idealized Algol is decidable*, in Proc. of 17th annual IEEE Symposium on Logic in Computer Science (LICS'02), Copenhagen, pages 245-256, Computer Society Press (2002).

[93] Paolini, L., *A stable programming language*, Information and Computation **204** (2006), 339-375.

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

[95] Reddy, U., *Global state considered unnecessary: introduction to object-based semantics*, in O'Hearn and Tennent (eds.), Algol-like languages, volume 2, Progress in Theroetical Computer Science, 227-295 (1997), Birkhäuser.

[96] Scott, D., *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Theoretical Computer Science **121**, 411-440 (1993) (manuscript circulated since 1969).

[97] Sieber, K., *Reasoning about sequential functions via logical relations*, Proc. Applications of Categories in Computer Science, Cambridge University Press (1992).

[98] Stoughton, A., *Mechanizing logical relations*, Proc. Mathematical Foundations of Progamming Semantics, Springer Lect. Notes in Comp. Sci. 802 (1994).

[99] Wadsworth, C., *The relation between computational and denotational properties for Scott's D-infinity models of the lambda-calculus*, SIAM J.ournal of Computing **5** (1976), 488-521.