Game semantics approach to higher-order complexity

Hugo Férée[a]

[a] *Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France*
Inria, Villers-lès-Nancy, F-54600, France
CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54506, France

Abstract

Game semantics was initially defined and used to characterize PCF functionals. We use this approach to propose a definition of complexity for such higher-order functions, as well as a class of polynomial time computable higher-order functions.

Keywords: higher-order, complexity, game semantics, BFF, PCF

1. Introduction

The complexity of first-order functions (*i.e.* over countable sets like integers, binary words, lists, graphs, etc.) has been defined and characterized in numerous ways. It is also the case to a lesser extent for second-order functions (*i.e.* functions defined over first-order functions), with an analogue of polynomial time defined by Mehlhorn [1] and characterized in particular by Kapron and Cook [2] using the oracle Turing machine model. In particular, this has applications in computable analysis where the complexity of real numbers, real functions and real operators has also been studied [3, 4, 5]. However, only few attempts have been made for larger spaces (mainly functions of order 3 or more). Most of them [6, 7, 8] are based on the study of the class BFF of Basic Feasible Functionals, which attempts to approach at higher-order what polynomial time complexity is for order 1. As underlined in [9], this class misses some intuitively feasible functionals and does not provide a general notion of running time and complexity. The difficulties appearing in defining a higher-order class of feasible functionals are in particular discussed in [10]. Other solutions include representing higher-order functions with first-order functions, such as Kleene associates [11], but the notion of complexity they naturally imply is not relevant. Our approach also consists in representing objects and functions as first-order functions, but with more structure. Game semantics allows this by describing functions as strategies, which can be seen as interactive analogues of Kleene associates.

A natural way to define complexity is to have a computational model with a notion of running time and of size for the inputs. The complexity is then defined as a bound on the running time given a bound on the size of the inputs. We will define a computational model which accesses its inputs using a game

dialogue. The inputs being strategies in some sequential game, we will need to define the size of such object, from which the notion of complexity will derive.

We will first provide some background for second-order functions and BFF in section 2 and provide in section 3 our presentation of games for higher-order functions, which were originally defined by Nickau [12] and Hyland and Ong [13] to solve the full abstraction problem for the programming language PCF presented by Plotkin [14].

Then in section 4, we will propose our approach for higher-order complexity, based on the previously defined games. We first give in subsection 4.1 a definition of size for strategies in such games, which is one of the main ingredients for the forthcoming definition of complexity. Then we describe two equivalent notions of complexity for these strategies (and thus for the functions they represent) based on this notion of size: the first approach (subsection 4.2) is based only on first-order complexity, whereas in the second one (subsection 4.3) we define a machine model based on oracle Turing machines, which is better adapted to the notion of game. Finally, subsection 4.4 defines a class of higher-order polynomial time computable PCF functions and states several results highlighting the relevance of this class.

Finally, in section 5, we underline the elements in game semantics which were needed for these definitions, and explain how they could be generalized to other kinds of games.

2. Background

Throughout this paper, we will be referring to first-order functions as functions of type $\mathbb{N}^k \to \mathbb{N}$ $(k \in \mathbb{N})$ and second-order functions as functions of type $(\prod_{i\leq j}(\mathbb{N}^{k_i} \to \mathbb{N}))\times\mathbb{N}^l \to \mathbb{N}$ $(k, i_1, \ldots, i_j, l \in \mathbb{N})$ and in particular to their respective corresponding polynomial-time computable classes (FPTIME and FPTIME_2). We often work out details for just the corresponding pure types $\mathbb{N} \to \mathbb{N}$ and $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$.

Second-order computability can be defined using the oracle Turing machine model, *i.e.* a machine which can evaluate its first-order input by dynamically querying its values at a given points. First-order complexity can be defined as a bound on the the running time of a Turing machine given a bound on the size of the input. In order to provide an analogue definition for second-order functions, Kapron and Cook [2] have defined the size of the inputs of oracle Turing machines, *i.e.* first-order functions.

Definition 1 (Size of a first-order function). Let f be a first-order function on finite words, *i.e.* of type $\Sigma^* \to \Sigma^*$. Its ***size*** $|f|_1 : \mathbb{N} \to \mathbb{N}$ is defined by:

$$\forall n \in \mathbb{N}, |f|_1(n) = \max_{|w|\leq n} |f(w)|, \tag{1}$$

where $|w|$ denotes the size of the binary word w. By abuse of notation, we also define the size of a first order function $g : \mathbb{N} \to \mathbb{N}$ on integers as the size $|f|_1$ of a function $f : \Sigma^* \to \Sigma^*$ on binary words encoding g in this sense:

$\forall n \in \mathbb{N}, bin(f(n)) = g(bin(n))$, where bin is the standard binary encoding of integers into binary numbers.

Together with the oracle Turing machine model, this notion allowed them to define second-order complexity. Note that a complexity bound is now a second-order function (just as well as a complexity bound for a first-order functions is itself a first-order function). They also have defined a second-order polynomial time complexity class, (which we will denote by FPTIME_2) using second-order polynomials, *i.e.* polynomials with first-order variables.

This notion is quite robust, since it extends nicely FPTIME (the restriction of FPTIME_2 to first-order function is FPTIME and it is stable by composition) and the original definition from Mehlhorn is an analogue of Cobham's [15] characterization of FPTIME using a function algebra.

The generalization of this function algebra by Cook and Urquhart [16] to all finite types, called PV^ω, is defined by the closure of FPTIME by several basic operators (composition, expansion, application) and a second-order bounded recursion on notation:

$$\mathcal{R}(x_0, F, B, x) = \begin{cases} x_0 \text{ if } x = 0 \\ t \text{ if } x > 0 \text{ and } |t| \leq B(x) \quad \text{where } t = F(x, \mathcal{R}(x_0, F, B, \lfloor \frac{x}{2} \rfloor)). \\ B(x) \text{ otherwise,} \end{cases}$$

The functions defined by this algebra are called the Basic Feasible Functionals (BFF). This class is also robust and all its elements are intuitively feasible, in the sense that they should all belong to an analogue of FPTIME and FPTIME_2 at higher types.

However, this class is not complete with respect to this criterion. Irwin, Kapron and Royer [8] provide an example of third-order function which is intuitively feasible but is not in BFF.

Example 1. *Let $f_x(y) = 1$ if $y = 2^x$ and 0 otherwise. The type 3 function $\Phi : ((\mathbb{N} \to \mathbb{N}) \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$ is defined by*

$$\Phi(G, x) = \begin{cases} 0 & \text{if } G(f_x) = G(\lambda y.0) \\ 1 & \text{otherwise} \end{cases}$$

is basic feasible: it can be computed by applying F_Φ to two polynomial time computable functions (f_x being polynomial time computable with respect to $|x|$) and comparing the results. However, the function Ψ defined by

$$\Psi(G, x) = 2^x \cdot \Phi(G, x) = \begin{cases} 0 & \text{if } G(f_x) = G(\lambda y.0) \\ 2^x & \text{otherwise.} \end{cases}$$

turns out not to be in BFF. The intuition is that in the second case, the size of the output is exponential in the size of x. However, one argument to say that it is feasible is that if we fall into the second case, it means that G was able to

distinguish between f_x and $\lambda y.0$. But these functions only differ at 2^x, which means that G has "queried" its inputs at this point, and since writing down this query takes $|2^x|$ steps, F_Φ's computation must be at least this long. From this point of view, the time needed to compute $\Psi(G, x)$ is at most twice the time to compute $\Phi(G, x)$, thus if Φ is considered feasible, so should Ψ.

In addition, BFF provides only a single complexity class and not a general notion of complexity (which would allow us to define exponential time, space or non-deterministic complexity classes for example).

One of the main missing elements for a higher-order notion of complexity is the notion of size. Indeed, the generalization of definition 1 to all finite types is not meaningful. It would only bound the size of the values taken by the function and not the amount of information it requires on its input to provide an answer, in other words its modulus of continuity (see definition 14). For instance, in the Ψ function from example 1, the modulus of the functions F should be taken into account: if it has a big modulus of continuity, then a computational model should be given more computation time.

In the following, we propose a solution to this problem, by seeing a computation as a dialogue between the function and its inputs, allowing to formalize claims such as: F "has to evaluate its input at 2^x". Note that Nickau [12] already mentioned that game semantics was probably a good model for complexity. Also, Buss and Kapron [17] have used this idea of dialogue in the context of second-order complexity. In particular, they define the *length of a dialogue*, which is really close to our definition of *size of a play* (see subsection 4.1). Finally, Royer [18] has used this approach to express the infeasibility of a higher-order functional, without using a generic notion of complexity though.

3. PCF-games

Higher order computations have been defined in many, often incomparable ways, but rarely with complexity in mind (see [19] for a quite exhaustive survey). Game semantics was initially used to provide a fully abstract semantics to PCF by Nickau [12] and Hyland and Ong [13] independently. It can be described as a way to represent computational processes as the confrontation of players in a game, like a program against its environment or parameters. We give here our own presentation of these games, which we will call PCF-*games*, which is inspired from the original ones as well as Gabbay and Ghica's nominal formalism [20], with effectivity in mind.

3.1. Arenas

The first element of this definition, namely the notion of *arena*, describes the possible actions (called *moves*) for each player, as well as rules, which define which moves can be played in a given context.

Definition 2 (Arena). An ***arena*** $\mathcal{A} = (\mathfrak{Q}, \mathfrak{A}, \lambda, \vdash, \mathfrak{I})$ is defined by:

- two disjoint sets $\mathfrak{Q}$ (***questions***) and $\mathfrak{A}$ (***answers***) which represents the set $\mathfrak{M} = \mathfrak{Q} \cup \mathfrak{A}$ of ***moves***;
- a ***polarity*** function $\lambda : \mathfrak{M} \to \{O, P\}$. Then, every move m is associated with the ***player*** (if $\lambda(m) = P$) or with the ***opponent*** (if $\lambda(m) = O$). We write $O^\perp = P$ and $P^\perp = O$ and if $A \subseteq \mathfrak{M}$, we will denote by A^P (resp., A^O) the set of moves in A belonging to the player (resp., the opponent);
- an ***enabling relation*** $\vdash \subseteq \mathfrak{Q} \times \mathfrak{M}$ which defines a graph structure over moves. If $m \vdash m'$ we say that m ***enables*** m' and a move can be enabled only by a question belonging to the other player:

  $$q \vdash m \implies \lambda(q) = \lambda(m)^\perp.$$

  We will also denote by $\vdash^*$ the transitive closure of $\vdash$ and we will say that m recursively enables m' if $m \vdash^* m'$.
- a set of ***initial questions*** $\mathfrak{I} \subseteq \mathfrak{Q}^O$ owned by the opponent and which are not enabled by any other move. We will furthermore call $\mathfrak{F}$ the set of ***final answers***, *i.e.* answers enabled by initial questions:

  $$\mathfrak{F} = \{a \in \mathfrak{A}^P \mid \exists i \in \mathfrak{I}, i \vdash a\}.$$

Intuitively, initial questions will be able so start a dialogue, whereas final answers will end it.

In the following, m will implicitly represent a move, q a question, a an answer, and i an initial question.

The PCF functions have finite types over natural numbers, *i.e.* of the form:

$$\tau ::= \tau \to \tau \mid \beta$$

where β denotes one of the two base types ι and o which will be interpreted as the set $\mathbb{N}$ of integers, and $\mathbb{B}$ of booleans. In the following, we may confuse these base types with their corresponding sets.

When defining with complexity, we don't want to deal with partial evaluation. For example, it is common to define the *max* function and its complexity as functions with two arguments (*i.e.* of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$) rather than as functions from integers to functions (*i.e.* of type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$). This is why we will represent these simple types in an uncurried style, with the following notation:

$$\tau_1 \times \cdots \times \tau_n \to \beta = \tau_1 \to (\cdots \to (\tau_n \to \beta) \dots).$$

Every finite type can indeed be described as a type of this shape, and the base types are in particular obtained with $n = 0$.

We may mostly focus on the base type ι as cases for o are often special cases for ι.

We will call the ***order*** of a type its maximal number of nested exponentials. In the type $\tau_1 \times \cdots \times \tau_n \to \beta$, the types $\tau_1, \ldots \tau_n$ will be called its ***argument-types***. In examples, we will mostly focus on *pure types*, *i.e.* types with only one (pure) argument-type.

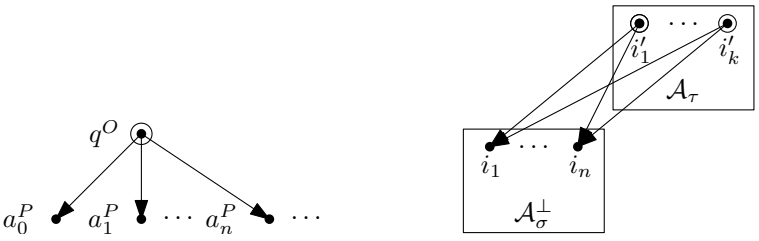
We will define games for each of these finite types, so we first need to define the arena $\mathcal{A}_\tau$ associated to a finite type τ by induction on τ.

Definition 3 (Arenas for base types). Figure 1a shows a representation of the arena for natural numbers: the exponents indicate the polarity, the arrows the enabling relation, and the circled moves are the initial questions.

Intuitively, a player representing a number n will answer r_n whenever its opponent asks him its value (i.e. by playing q).

The arena $\mathcal{A}_\mathbb{B}$ for base type o is similar to $\mathcal{A}_\mathbb{N}$ but only has two answers, representing the two boolean values.

Then, we recall the definitions (from [20] for example) for the exponential arena from two given arenas as described in figure 1b.



(a) The arena $\mathcal{A}_\mathbb{N}$ associated with the base type ι.

(b) The arena $\mathcal{A}_{\sigma\to\tau}$ constructed from the arenas $\mathcal{A}_\sigma$ and $\mathcal{A}_\tau$.

Definition 4 (Exponential arena). The arena $\mathcal{A}_{\sigma\to\tau}$ is also built from the disjoint union of $\mathcal{A}_\sigma$ and $\mathcal{A}_\tau$. Its initial questions are those which were initial in $\mathcal{A}_\tau$ and they enable the initial questions in $\mathcal{A}_\sigma$. Moreover, the polarity of the moves in $\mathcal{A}_\sigma$ is reversed. This construction is represented in figure 1b, where $\mathcal{A}^\perp$ represents the arena $\mathcal{A}$ whose polarities have been reversed. More precisely: $\mathcal{A}_{\sigma\to\tau} = (\mathfrak{Q}_\sigma \sqcup \mathfrak{Q}_\tau, \mathfrak{A}_\sigma \sqcup \mathfrak{A}_\tau, \lambda_\sigma^\perp \amalg \lambda_\tau, \vdash_\sigma \cup \vdash_\tau \cup (\mathfrak{I}_\tau \times \mathfrak{I}_\sigma), \mathfrak{I}_\tau)$.

The arenas built this way for each finite type are finite trees whose depth is the order of the type.

We can now have a look at the first simple types and see how their arenas can be seen as way to present the possible interactions between existing computational models (mainly machine models) and their inputs.

Example 2 (Arena $\mathcal{A}_{\mathbb{N}\to\mathbb{N}}$)**.** *The arena $\mathcal{A}_{\mathbb{N}\to\mathbb{N}}$ (figure 2), represents the structure of functions of type $\iota \to \iota$. It can be seen as a way to describe the interactions in a Turing machine: when the machine is executed (i.e. the opponent*

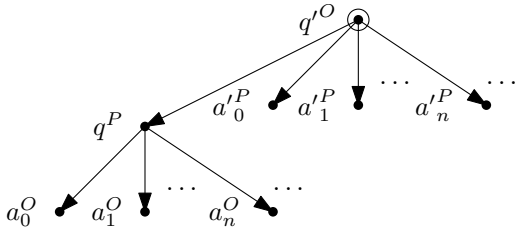


Figure 2: Representation of arena $\mathcal{A}_{\mathbb{N}\to\mathbb{N}}$.

asks for its value by playing q'), the machine implicitly asks (question q) the value (answer a_n) of its input, which is then written to the input tape. The machine can then answer to the initial question, by providing the result on the output tape (i.e. playing some answer $a'_{n'}$). The enabling relation (represented in the figure by the arrows) express the constraints on the possible actions of the machine and its oracle: the machine can only terminate or do an oracle call after it has been started (arrows starting from q'), and the oracle can provide an answer only if it has been queried by the machine (arrows starting from q).

Example 3 (Arena $\mathcal{A}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$)**.** *Similarly, we can go one order higher and define the arena $\mathcal{A}_{(\iota\to\iota)\to\iota}$ (figure 3). It can be seen as a way to represent the interactions between an oracle Turing machine and its input in the following way. When started (question q''), the machine can do several oracle calls, i.e. play a question q', and answer some a_n to an implicit question q from the oracle (where n is the oracle query). Then the oracle can answer the question q', i.e. give some value a'_k (the oracle answer) to the machine. This is either repeated indefinitely, or the machine eventually outputs a result after a finite number of such oracle calls, i.e. answers to the initial question with an move a''_l, which is the result of the computation.*

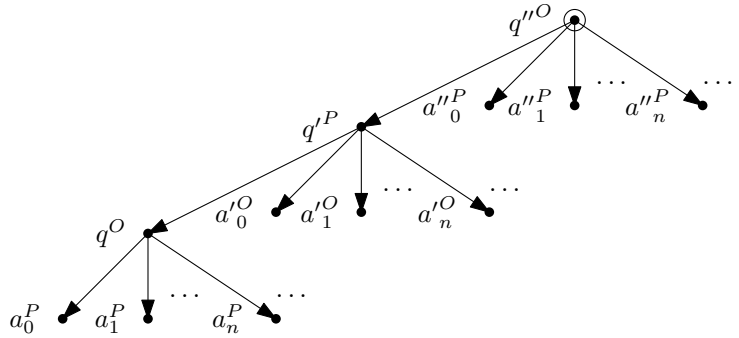


Figure 3: Representation of arena $\mathcal{A}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$.

3.2. Plays

A game will not only be characterized by its possible moves, but by the allowed successions of moves called plays. As moves will have to be enabled by other moves, we will make this reference to another move explicit using names.

Definition 5 (Plays). Given an arena $\mathcal{A}$, a ***named move*** is a move m associated with an integer α denoted this way: $m[\alpha]$. A *play* is then a finite sequence of named moves. As in nominal game semantics (of which our presentation is only a simplified version), we will call α the ***justifying name*** of the named move.

We may write $m[\alpha] \in p$ if this named move is in the play p and we say that a named move $m[\alpha]$ ***justifies*** a further named move $m'[\beta]$ in the play p if $m[\alpha]$ is in position number β in p and m' is enabled by m in the arena.

In the following, $\alpha, \beta, \dots$ will implicitly denote names, and $p, p' \dots$ plays. We may also use moves and named moves indifferently when the value of the name is not relevant. We will also denote by p_α the move (or named move) in position α in p.

In addition, we will say that a question $q[\alpha]$ in a play p is ***closed*** if it justifies a further answer a in p.

In this case, we say that a answers this question and if no such a exists, we will say that q is ***open*** in p.

In order to define a game, we need to describe the set of allowed plays, which can be done by imposing some conditions on plays. The following rules will be useful to define PCF-games.

Definition 6. If p is a play in an arena $\mathcal{A}$, we say that p is

- ***justified*** if every non-initial move is enabled by a previous move in p:

$$\forall m, p', \alpha, m \notin \mathfrak{I} \wedge p = p'm[\alpha] \cdots \implies p'_\alpha \vdash m;$$

- ***well-opened*** if contains only one initial move, at the beginning of p ;
- ***alternating*** if any two consecutive moves belong to different protagonists:

$$\forall p, \alpha, m', \beta, p = \dots m[\alpha], m'[\beta] \cdots \implies \lambda(m) = \lambda(m')^\perp;$$

- ***strictly scoped*** if answering a question prevents further moves to be justified by this question. In particular, a question cannot be answered twice ;
- ***strictly nested*** if pairs formed by questions and their corresponding answers form a valid bracketing.

Remark 1. *In justified plays, initial moves do not need to be justified, so the value of their justification name is not relevant and we may omit it.*

Remark 2. *In a justified alternating play, the polarity of the last move is characterized by the parity of the number of moves. In this case, we will call a player (resp. opponent) play any such play ending with a player (resp. opponent) move and if* $\mathbb{P}$ *is a set of plays, we will denote by* $\mathbb{P}^P$ *(resp.* $\mathbb{P}^O$*) the corresponding subset.*

Remark 3. *This is a particular application of nominal game semantics, where each move in a play is implicitly denoted by its position. Note that more clever naming systems could be used, especially if they manage to use smaller names, for example by reusing names when they can no longer be used (which is the case when the* strictly scoped *rule is implied). This idea can be seen as dynamical memory management, and the reader can refer to [21] for a study on the issue of name reuse. However, such a choice is not restrictive for our further definition of complexity.*

3.3. Innocent strategies

We can now define ***strategies***, that is deterministic decisions taken by a player, given its current knowledge of the play.

Definition 7 (Strategy). Given an arena and a set $\mathbb{P}$ of plays, a strategy s is a partial function from a subset $dom(s)$ of valid plays ($\mathbb{P}$), produces a named move which extends it into a valid play:

$$\forall p \in dom(s), (p :: s(p)) \in \mathbb{P},$$

where :: is the finite sequence extension operator defined by:

$$(m_1, \ldots, m_n) :: m_{n+1} = (m_1, \ldots, m_n, m_{n+1}).$$

Now, one of the major characteristics of the strategies used for PCF functions is their *innocence* property: their output should depend only on partial knowledge on the input play, called *view*.

We remind the notion of *innocent strategy* which was independently introduced by Hyland and Ong [13] and Nickau [22].

Definition 8. Let $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$ be a finite type. If p is an opponent play in $\mathcal{A}_\tau$ satisfying the properties given in definition 6, then its view $\text{view}(p)$ can be defined this way, starting from the end of p:

- if we meet an open player's question q justified by a player (resp. opponent) question q', then all moves between q' and q are removed from p and all further justifying names are modified accordingly;
- if we meet a player's answer a answering an opponent (player) question q, then all moves between q and a are removed from p and all further justifying names are modified accordingly;
- in both cases, we continue this process with the move placed to the left of q.

If the play indeed satisfies the aforementioned rules, we can easily see that we stop when meeting the initial move and that view(p) is an opponent play still satisfying these rules.

An ***innocent strategy*** s is a strategy which depends only on the current view of the play:

$$\forall p \in dom(s), s(p) = s(\text{view}(p))$$

Intuitively, it implies that if the opponent has answered a player's question, then the player can only take into account the answer and not the intermediary dialogue which was used to obtain this answer.

We now have all the necessary definitions to define PCF-games.

Definition 9 (PCF-game). For every finite type τ, the corresponding PCF-game $\mathcal{G}_\tau$ is given by the set $\mathbb{S}_\tau$ of *innocent* strategies over *alternating strictly-scoped strictly-nested well-opened* plays whose answers are justified by last open question, over the arena $\mathcal{A}_\tau$.

In the following, we will always implicitly restrict to this definition ; in particular, every strategy will be innocent and restricted to the previously defined set of valid plays.

Also, if $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$, then $\mathcal{G}_{\tau_1}, \ldots, \mathcal{G}_{\tau_n}$ will be called the ***sub-games*** of $\mathcal{G}_\tau$ and strategies in these games will be called ***argument strategies*** for any strategy in $\mathcal{G}_\tau$.

3.4. Confrontation

We can now define the confrontation of a strategy of an exponential type against strategies for each of its argument-types. This will allow us to relate these strategies with functions of the corresponding type.

First, notice that a play in the arena of a given type $\tau_1 \times \cdots \times \tau_n \to \beta$ is mostly composed of plays in the arenas of its argument-types $(\tau_1, \ldots, \tau_n)$.

Indeed, given the definition of arenas for finite types (definitions 3, and 4), the moves enabled by the initial move are either final answers or questions which are initial in one of the arenas $\mathcal{A}^\perp_{\tau_j}$ composing $\mathcal{A}_\tau$. A well-opened justified play p in $\mathcal{A}_\tau$ can thus be seen as an initial move followed by intricate well-opened justified plays in some of the arenas $\mathcal{A}_{\tau_j}$ (possibly several for each arena), possibly ended by a final answer. If in addition this play is a player play not ending with a final answer, then its last move m belongs to some $\mathcal{A}_{\tau_j}$ for some j. In this case, m is recursively justified by a question i_j which is initial in $\mathcal{A}_{\tau_j}$ and we can define a new play p' in $\mathcal{A}_{\tau_j}$ (called the current ***sub-play*** of p) by selecting all the moved in p which are recursively justified by i_j and by changing their justification names accordingly. In addition, this play verifies the rules of definition 6 as soon as p does.

Conversely, if such a sub-play is extended by a additional move $m'[\alpha]$ in $\mathcal{A}_{\tau_j}$, p can itself be extended by a move $m'[\beta]$ where β points to the same move in p as α does in p'. Once again, the rules of the game are verified in this new play as soon as they were in p and in $p' :: m'$.

These constructions (which can be easily carried out in polynomial time), are key tools for a strategy in interacting with *argument strategies* (*i.e.* strategies for argument-types).

Definition 10 (Confrontation). Let $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$ be a finite type. Given a strategy s in $\mathbb{S}_\tau$, and a strategy s_j in each $\mathbb{S}_{\tau_j}$, the ***confrontation*** between s and the s_j is the process which incrementally builds a valid play p in $\mathcal{G}_\tau$ in the following way, where p is considered as a variable, and the symbol $\leftarrow$ denotes imperative-style assignment.

The play begins with the only initial question i in $\mathcal{A}_\tau$: $p \leftarrow i$. Then, the strategy s plays alternatively against one of the s_j as follows.

If the current play is p, then s plays by adding the named move $s(p)$ to p: $p \leftarrow p :: s(p)$. If this move is of the form $a[0]$, *i.e.* if s has answered the initial question, then the confrontation ends. Otherwise, the move $s(p)$ belongs to the arena $\mathcal{A}_{\tau_j}$ for some j. Then, if $s_j(p_j)$ is defined (where p_j is the current sub-play), it is converted to a named opponent move in $\mathcal{G}_\tau$ and added to the end of p.

Given a strategy $s \in \mathbb{S}_\tau$, this operation defines a partial function from strategies in sub-games to final moves: $s[] : \mathbb{S}_{\tau_1} \times \cdots \times \mathbb{S}_{\tau_n} \hookrightarrow \mathfrak{F}$: if s ends its confrontation against s_i with the final answer a, then $s[s_1, \ldots, s_n] = a$ and a is called the ***result of the confrontation*** and if no such a exists, then $s[s_1, \ldots, s_n]$ is undefined.

Finally, the complete play p, when it exists, is called the ***history of the confrontation*** and we will denote it by $H(s, s_1, \ldots, s_n)$.

Remark 4. *There are two ways a confrontation can fail to terminate: either one of the strategies is undefined on its current play, or the dialogue goes on indefinitely.*

Remark 5. *Our notion of confrontation is nothing new. Indeed, Hyland and Ong define very similarly a* strategy composition*, which from a strategy for type $\tau \to \sigma$ and a strategy for type τ defined a strategy for type σ.*

Thus $s[s_1, \ldots, s_n]$ is (almost) equal to the successive compositions of s with $s_1, \ldots, s_n$, i.e. until we obtain a strategy over a base type. In other words, confrontation is to full application what strategy composition is to partial application.

Now we can make a link between strategies in these games and higher type functions. We will consider partial functions, and $\perp$ will denote the undefined value.

Definition 11. A strategy s in $\mathbb{S}_\tau$ ***represents*** a partial function of type τ in this sense: if $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$, then s represents a partial function $F : \tau$ if for all strategies $s_1, \ldots, s_n$ representing functions $f_1 : \tau_1, \ldots, f_n : \tau_n$, F is well defined on $(f_1, \ldots, f_n)$ if and only if the confrontation between s and $s_1, \ldots, s_n$ terminates, and in this case the final answer encodes $F(f_1, \ldots, f_n)$ (*i.e.* it is the final answer of the form $a_{F(f_1,\ldots,f_n)}$).

Example 4.

- *On the play q composed of only the initial question, a strategy in $\mathbb{S}_{\mathbb{N}}$ (resp. $\mathbb{S}_{\mathbb{B}}$) is either undefined (and thus represents the undefined element) or equal to some final answer $a_n[0]$ for some $n \in \mathbb{N}$ (resp. $n \in \mathbb{B}$) and thus represents the integer (resp. boolean) n.*

- *The confrontation of a strategy s in $\mathbb{S}_{\mathbb{N}\to\mathbb{N}}$ against a strategy s_n in $\mathbb{S}_{\mathbb{N}}$ which represents an integer n can fail to terminate if s indefinitely asks the question q or if it is undefined at some point. The function $s[]$ defines a partial function f of type $\mathbb{N} \to \mathbb{N}$ defined by $\forall n \in \mathbb{N}, s[s_n] = a'_{f(n)}$. If the confrontation terminates, then it is of the form:*

$$q', q[0], a_n[1], \ldots, q[0], a_n[2 * k + 1], a'_{f(n)}[0]$$

- *Similarly, a strategy s_F in $\mathbb{S}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$ defines a partial second-order function F in the sense that if s_f represents a partial first-order function (as defined above), then $s_F[s_f] = a''_{F(f)}$. In this case, the history of the confrontation is of the form:*

$$q'', p_1, \ldots, p_n, a''_{F(f)}$$

where the p_i are plays similar to the one in the previous point (with shifted justifications), i.e. even prefixes of plays of the form:

$$q'[0], q[k_i]a_{x_i}[k_i + 1], \ldots, a'_{f(x_i)}[k_i]$$

This history can be seen as the description of the behavior of an oracle Turing machine computing F on oracle f, where $x_1, \ldots x_n$ denote the oracle calls (and $f(x_1), \ldots f(x_n)$ the corresponding oracle answers).

3.5. Computability

In order to define computable strategies, we need an encoding for named moves and plays in PCF-games. There is indeed, for a given type τ, a natural binary encoding of moves in $\mathcal{A}_\tau$: there are a finite number of questions which are thus encoded by words of given length; and there are only a finite number of answers representing the integer n ($a_n, a'_n, \ldots$ in the examples) which can then be encoded by words of length $\mathcal{O}(\log_2(n))$. Finally, justifying names can be encoded using the usual binary encoding of integers and thus a basic encoding of plays also follows and will be denoted by **enc** in the following.

Of course, different encodings (over the same game) can induce different notions of computability and complexity, in the same way that using unary rather than binary representation of natural numbers leads to a different notion of first-order complexity. However, any canonical encoding such as the previous one would lead to the same notion of computability and complexity for large enough complexity classes (especially the polynomial time class which we will define later). A more clever encoding will only be necessary to study sub-linear complexity classes, in which case we will also require applying remark 3 which we will not do here for simplicity.

Such an encoding allows us to see a strategy as a first-order function on binary words.

Definition 12 (Computable strategy). A strategy s is computable if there exists a computable function $f_s : \Sigma^* \to \Sigma^*$ on binary words which maps the encoding of every play in the domain of s to the encoding of its image by s:

$$\forall p \in dom(s), f_s(\mathbf{enc}(p)) = \mathbf{enc}(s(p)).$$

By extension, we will say that a higher-order function is computable if it is represented by a computable strategy. It is clear that the computable strategies in the games $\mathbb{S}_{\mathbb{N}\to\mathbb{N}}$ and $\mathbb{S}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$ (see example 4) represent exactly the usual partial sequential computable functions of order 1 and order 2 respectively.

Remark 6. *Given s, $s[\cdot]$ can be seen as a second-order function on binary words. Moreover, if s is computable in the sense of definition 12, then $s[\cdot]$ can be seen as a (partial) computable second-order functional. Indeed, we can build an oracle Turing machine from a Turing machine computing f_s, which on oracles representing functions $f_{s_1}, \ldots, f_{s_n}$ internally computes the history of the game by simulating the process of confrontation: it iteratively computes the next move to play ($f_s(p)$) and add it to the current play (initially containing the initial question), then stops if it is a final answer (and output it), or queries the right oracle (depending on $s(p)$) on its current knowledge of the play.*

In terms of computability, these games capture exactly PCF.

Theorem 1 ([22, 13])**.** *The partial functions represented by computable strategies in* PCF*-games are exactly the* PCF*-computable functionals.*

4. Complexity

We will now define a notion of complexity for strategies in our games. For this, as explained earlier, we will need to provide a notion of size for the "inputs" of such strategies, which are themselves strategies for PCF-games. Then we will define their complexity using two distinct approaches and finally propose a notion of polynomial time complexity for such strategies and, by extension, for the functions they represent.

4.1. Size of a strategy

We define the ***size of a play*** as the size of its binary encoding and also denote it $|\cdot|$ by extension.

Note that, this is polynomially equivalent to the sum of the sizes of its moves or named moves given our choice for names (as positions). This remark shows that, as stated earlier, this choice was not really critical, and any other reasonable choice could have been made, as soon as we are interested in complexity classes closed by polynomial composition.

We can now define the size of a strategy s in $\mathbb{S}_\tau$ as a function of type τ together with a partial order $\preccurlyeq_\tau$ on such functions. Intuitively, given a bound on the size of a strategy in a sub-game, it bounds the size of the confrontation of s against this strategy.

For the games previously used as examples, this definition of size of strategies is related to the size of the function they compute. In particular, as we will see in proposition 1, the size of a strategy in $\mathbb{S}_{\mathbb{N}}$ is roughly the binary size of the integer it represents, and the order $\preccurlyeq_{\mathbb{N}}$ is defined by comparing the size of the binary representation of integers. Similarly, the size of the strategy representing a boolean number has constant size and $\preccurlyeq_{\mathbb{B}}$ is the trivial equality order.

Definition 13 (Size of a strategy). The size function $\|\cdot\|_\tau : \mathbb{S}_\tau \to \tau$ of strategies in $\mathbb{S}_\tau$ as well as partial order $\preccurlyeq_\tau$ on functions of type τ are defined by induction on τ. Note that the base cases are handled by the following general case with $n = 0$.

If $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$, and $s \in \mathbb{S}_\tau$ then:

$$\|s\|_\tau = \lambda b_1 \ldots \lambda b_n. \sup\{|H(s, s_1, \ldots s_n)| \ : \ \forall 1 \leq i \leq n, s_i \in \mathbb{S}_i \wedge \|s_i\|_{\tau_i} \preccurlyeq_{\tau_i} b_i\}$$

i.e. given input bounds $b_1, \ldots, b_n$, it is the maximal size of the history of the confrontation of s against strategies whose size is smaller than these bounds.

Additionally, for all $F, B : \tau$, we say that F is smaller than B ($F \preccurlyeq_\tau B$) if:

$$\begin{aligned}\forall b_1, \ldots, b_n, \forall s_1, \ldots s_n, \|s_1\|_{\tau_1} \preccurlyeq_{\tau_1} b_1 \wedge \cdots \wedge \|s_n\|_{\tau_n} \preccurlyeq_{\tau_n} b_n \implies \\ F(\|s_1\|_{\tau_1}, \ldots, \|s_n\|_{\tau_n}) \leq B(b_1, \ldots, b_n)\end{aligned}$$

Proposition 1. *In the game $\mathcal{G}_{\mathbb{N}}$ (which is the base case), every integer n has a unique strategy whose size is bounded by $c_0 \cdot |n|$, for a constant c_0. Conversely, any strategy of size k represents an integer whose binary size is smaller than k.*

Proof. The confrontation of a strategy representing an integer n against no strategy ($n = 0$) leads to a history $q, a_n[0]$, whose encoding has a binary size roughly bounded by the binary size of n.

The converse is immediate, since the confrontation of any bounded-size strategy terminates, *i.e.* ends with an answer a_n with $|n| \leq k$. □

Similarly, a first-order function f has a strategy whose size is roughly the size of f, *i.e.* $|f|_1$, as defined by Kapron and Cook (see definition 1). In other words, the size of such a strategy bounds the size of its output given a bound on the size of its input.

Proposition 2. *Every total function $f : \mathbb{N} \to \mathbb{N}$ has a strategy in $\mathbb{S}_{\mathbb{N}\to\mathbb{N}}$ whose size is bounded by $\lambda k.c + k + |f|_1(k)$ for a constant c independent from f.*

Conversely, $|f|_1 \preccurlyeq_{\mathbb{N}\to\mathbb{N}} \lambda k.\|s_f\|_{\mathbb{N}\to\mathbb{N}}(c_0 \cdot k)$ for every strategy s_f representing f.

Proof. We consider the canonical strategy s_f, which queries its input x (*i.e.* plays $q[0]$ and gets an answer $a_x[1]$), and then answers the value of f on this input (*i.e.* plays $a'_{f(x)}[0]$). The confrontation of s_f against a strategy in $\mathbb{S}_{\mathbb{N}}$ of size bounded by k (thus representing an integer x of size smaller than k according to proposition 1) produces a history of the form $q', q, a_x, a'_{f(x)}$. The

size of this play is bounded by $c + |x| + |f(x)|$ for some constant c. Maximizing over all argument strategy of size bounded by k provides the first result.

For the second result, it is sufficient to notice that proposition 1 implies that $\{s \in \mathbb{S}_{\mathbb{N}} \mid \|s\|_{\mathbb{N}} \leq c_0 \cdot k\}$ contains all the strategies representing integers of size bounded by k. Their history against s_f then contains $a'_{f(n)}$ for an integer n of size bounded by k. Since $|a'_{f(n)}| \geq |f(n)|$, we have $\|s_f\|_{\mathbb{N}\to\mathbb{N}}(c_0 \cdot k) \geq \max_{|n|\leq k} |f(n)| = |f|_1(k)$. □

Note that there are strategies whose size is not bounded. However, strategies for first-order functions have bounded size if and only if they represent a total function (due to the definition of $\preccurlyeq$).

For strategies representing second-order functions, the notion of size takes into account the notion of modulus of continuity and thus not every function (not even total function) is represented by a strategy.

Definition 14 (Modulus of continuity). A total function $B : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is a modulus of continuity of $F : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ if:

$$\forall b, f, g, (|f|_1, |g|_1 \preccurlyeq_{\mathbb{N}\to\mathbb{N}} b \wedge \forall n \leq B(b), f(n) = g(n)) \implies F(f) = F(g).$$

Proposition 3. *If $F : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is represented by a strategy in $\mathbb{S}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$ of size bounded by $B : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, then $\lambda b.2^{B(\lambda k.c+k+b(k))}$ is a modulus of continuity for F (where c is a constant independent from F).*

Proof. Let $f, g, b : \mathbb{N} \to \mathbb{N}$ such that $|f|_1$ and $|g|_1$ are bounded by b and are equal on all inputs of size bounded by $2^{B(\lambda k.c+k+b(k))}$. According to proposition 2, f and g have strategies of size bounded by $\lambda k.c+k+b(k)$. The size of the history of the confrontation of the strategy of size B representing F against each of these two strategies is bounded by $B(\lambda k.c + k + b(k))$ by definition. Then, this history can not contain answers of size greater than $B(\lambda k.c + k + b(k))$. This implies that the strategy for F is not able to evaluate the values of f and g for inputs whose size are larger than this bound. Since it can not distinguish between these two strategies, then the histories are equal and in particular the results of the confrontations are equal, which implies that $F(f) = F(g)$. Then, $2^{B(\lambda k.c+k+b(k))}$ is indeed a modulus of continuity for F. □

The converse is also true: having a modulus of continuity implies having a strategy with bounded size.

Proposition 4. *If $F : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ has a modulus of continuity, then it is represented by a strategy whose size is bounded.*

Proof. If F is a function with modulus B we can define as strategy s for F this way: the strategy asks for the successive values of its argument (or opponent) from 0 up to the minimal $n \in \mathbb{N}$ such that F is constant on the set of functions with the same values on $\{0, \ldots, n\}$ and gives this constant as final answer.

Let s_f be a strategy computing a function $f : \mathbb{N} \to \mathbb{N}$. The history $H(s, s_f)$ is composed of n evaluations of f for some $n \in \mathbb{N}$. Each evaluation accounts for a

size of at most $\|s_f\|(c_0 \cdot |n|)$ according to proposition 1. Also, n is itself bounded by $B(|f|_1)$ by definition of the modulus, which itself can be bounded with respect to $\|s_f\|$ according to the second part of proposition 2 and by monotonicity of B. Finally, the size of the final answer (corresponding to $F(f)$) is bounded (up to a constant) by $\max_{|g|_1 \leq |f|_1} F(g)$, which itself can be expressed with $\|s_f\|$. The size of the whole history is then bounded with respect to the size of s_f, which allows us to conclude. Note that an explicit bound can be expressed, but it is of no use here. □

Together, these propositions allow us to characterize the second-order functions which can be represented by finite size strategies (*i.e.* in $\mathbb{S}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$).

Theorem 2. *A function of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is represented by a strategy in $\mathbb{S}_{(\mathbb{N}\to\mathbb{N})\to\mathbb{N}}$ with a well-defined size if and only if it is a continuous function defined on every total input.*

Remark 7. *Seen as a class of functions, the set of bounded size strategies can be seen as hereditarily total functions. Indeed, if the size of s is bounded, then for every bounded-size strategy s', the confrontation between s and s' terminates, and in particular, the size of the result is bounded by the composition of the sizes of the strategies $|s[s']| \leq \|s\|(\|s'\|)$.*

4.2. Accumulated complexity

The notion of size of the inputs was the main missing point in the general definition of complexity. Now that we have such a notion on strategies, we can define a first notion of complexity.

The first definition we can think of uses the representation of a strategy as a first-order function over binary words which allowed us to define the notion of computable strategies (definition 12). However, defining the complexity of a strategy as the complexity of the corresponding first-order function is not meaningful. Indeed, a strategy can force the dialogue to be as long as it wants, in particular until the time to compute a given move is polynomially bounded in the size of the current play (see example 5).

Example 5. *For all exponential time computable $f : \mathbb{N} \to \mathbb{N}$, there is a strategy s representing the second-order function $\lambda g.f(g(0))$ such that its associated first-order function is polynomial time computable.*

Indeed, the strategy s can query the successive values of its input g until the size of the current play is greater than $2^{|g(0)|}$, before giving the final answer corresponding to $f(g(0))$. Every computation made by s was polynomially bounded with respect to the size of its input (the current play), however, the length of the dialogue is exponential, which allows s to compute a high complexity function.

The following definition proposes to define the complexity as a measure of the time required to simulate the confrontation. To do so, it adds up the time needed to compute every move of the history. Note that, as well as the complexity of first and second-order functions is respectively a first or second-order function itself, the complexity of a strategy in $\mathbb{S}_\tau$ is a higher-order function of type τ.

Definition 15 (Accumulated complexity). A strategy s in a game $\mathcal{G}_\tau$ is computable in time $T : \tau$ if there is a function $T_0 : \mathbb{N} \to \mathbb{N}$ such that s is computable (as a first-order function on binary words) in time T_0, such that for all strategies $s_1, \dots, s_n$ in the sub-games of $\mathcal{G}_\tau$ with

$$H(s, s_1, \dots s_n) = m_1, m'_1, \dots m_t, m'_t,$$

we have

$$\sum_{1 \leq j \leq t} |m_j| + T_0(\mathbf{enc}(m_1, m'_1, \dots, m_j)) \leq T(\|s_1\|_{\tau_1}, \dots, \|s_n\|_{\tau_n}). \tag{2}$$

This is equivalent to say that the running time of the oracle Turing machine computing $s[]$ (described in remark 6) on oracle s_i (*i.e.* on the corresponding first-order representation) is bounded by $T(\|s_1\|_{\tau_1}, \dots, \|s_n\|_{\tau_n})$.

This definition measures the cost of the moves played by the opponent (similarly as in the definition of the running time of an oracle Turing machine) as well as the time needed by the strategy to play each of its own moves.

This definition is polynomially equivalent to the usual definitions on first and second-order functions and in particular it defines the same polynomial time complexity classes.

Proposition 5. FPTIME *(resp.* FPTIME_2*) is the set of functions of first-order (resp. second-order) type τ represented by a strategy in $\mathbb{S}_\tau$ whose accumulated complexity is bounded by a polynomial (resp. second-order polynomial).*

Proof. For first-order (resp. second-order) functions, this complexity measure is larger than the running time of a Turing machine (resp. oracle Turing machine) computing the function. Conversely, in the case of first-order functions, it exceeds it only by the sum of the sizes of the inputs, which keeps the complexity polynomial. In the case of second-order functions, each element of the sum is bounded by the total running time of an oracle Turing machine computing the function (plus the size of the possible first-order inputs) ; and t is bounded by the number of oracle calls (plus the number of first-order inputs), which itself is polynomially bounded in the size of the inputs, which makes it overall quadratically equivalent to the usual complexity. □

Already at order 1, the size (as defined by Kapron and Cook) of a function is a lower bound on its complexity, and this result can be generalized to strategies.

Proposition 6. *If s is a strategy in $\mathbb{S}_\tau$ whose accumulated complexity is bounded by B, then $\|s\|_\tau \preccurlyeq_\tau B$.*

Proof. This is immediate from the definition of accumulated complexity. Indeed, it bounds the sum of the sizes of the moves from the opponent, as well as the sum of the times to compute the player moves, which themselves bound the size of these moves. The accumulated complexity then bounds the sum of the sizes of the moves in the history given a bound on the size of the argument strategy, which is the definition of the size given in definition 13. □

Remark 8. *Proposition 6, together with remark 7, implies that the strategies with well-defined complexity are in particular defined on argument strategies which have well-defined complexity and this class of strategies is thus also hereditarily total in this sense.*

Conversely, the size function is in some sense the greatest lower bound on the complexity. More precisely, if we define the ***relativized accumulated complexity*** of a strategy s as the accumulated complexity of a machine computing s with the help of an arbitrary first-order oracle, then the following proposition holds.

Proposition 7. *Given a strategy in $\mathbb{S}_\tau$, its relativized accumulated complexity is bounded by $T : \tau$ if and only if its size is bounded by T.*

Proof. Proposition 6 is also true if the machine has access to an additional oracle, which proves the first implication.

Conversely, if $f_s : \Sigma^* \to \Sigma^*$ is the first-order function associated to $s \in \mathbb{S}_\tau$, then s is computed by a machine with oracle f_s and whose complexity is bounded by $T_0 = \lambda n.n$. Then, the left side of equation 2 is exactly the size of the history, which proves that the accumulated complexity is exactly the size of s. □

This proposition helps to understand these new notions of size and complexity. Indeed, it shows that a strategy can have a high complexity for two very different reasons: either it encodes a intrinsically difficult problem which requires a lot of *standard* computation time (represented by T_0 in definition 15), or it requires to know a large amount of information on its arguments (which web obtain when we remove the *standard* computation time by using an oracle), and this amount is what we defined as its size.

4.3. Game machines

The accumulated complexity corresponds to the usual notions of complexity at orders 1 and 2, which is a good sanity check. However, it may seem unnatural as it does not consist in a bound on the running time of a machine, but rather in the sum of several executions of the machine. In order to underline the validity of such a notion, we define a machine model which is more adapted to games and which can be seen as a generalization of the oracle Turing machine model. The notion of complexity induced by this model will be polynomially equivalent to the accumulated complexity.

Definition 16 (Game machine). Given a finite type $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$, a game machine on $\mathcal{G}_\tau$ is a kind of oracle Turing machine with one initial state and n oracles. Starting the machine in its initial state can be seen as starting a confrontation with the initial question. When the machine makes an oracle call, its special tape contains the encoding of a move m (we can say that the machine *plays* m). If it is the encoding of a final answer, then the computation stops and this answer represents the result of the computation. Otherwise, if this moves concerns some argument numbered i, then the corresponding oracle writes then encoding of a move on the tape and the computation goes on.

We will say that an oracle represents a strategy s_i (in $\mathbb{S}_{\tau_i}$) if for each oracle call, the oracle answer is the encoding of the named move m' obtained (as described in the definition of the confrontation) by applying s_i to the current sub-play and changing its justifying name accordingly.

Similarly, a machine is said to compute a strategy s if each of its oracle calls is the encoding of $m = s(p)$ where p is the current play, and stops on a final answer which encodes the result of the confrontation against the strategies given as oracle.

This model does not bring anything new in terms of computability.

Proposition 8. *A strategy is computable as a first-order function if and only if it is computed by a game machine.*

Proof. Given a game machine computing a strategy, we can build a standard Turing machine which computes the associated first-order function. Given the encoding of a play as input, it will simulate the game machine by replacing the oracle (playing the opponent strategy) by the list of opponent's moves in the play until all these moves have been played. Then, the machine outputs the next oracle call of the game machine, or its output if it terminates.

Conversely, we can build a game machine which stores the current play on a tape, and each time it must play (*i.e.* make an oracle call), it simulates a Turing machine computing s on this current play in order to get the right move to play. In other words, this machine is the oracle Turing machine computing the function $s[]$ in remark 6. □

Since the notion of running time of oracle Turing machines has already been defined, we can provide a complexity notion for this model too.

Definition 17 (Complexity of a strategy). The ***running time*** of a game machine is the same as the one of an oracle Turing machine: the cost of an oracle call is the size of the oracle answer.

Now we say that the complexity of a game machine on $\mathcal{G}_\tau$ is bounded by $T : \tau$ if for all strategies $s_1, \ldots, s_n$ with bounded sizes, the computation time of the game machine on the oracles representing $s_1, \ldots, s_n$ is bounded by $T(\|s_1\|_{\tau_1}, \ldots, \|s_n\|_{\tau_n})$.

From now on, we will say that the ***complexity of a strategy*** is bounded by T if there exists a game machine with running time T computing this strategy.

Remark 9. *It is easy to see that as for the accumulated complexity, the complexity of a strategy (in the sense of definition 17) bounds the size of the strategy.*

Moreover, the running time of a game machine on a given input strategy bounds the number of moves in the history.

We now prove that our two definitions of complexity are equivalent.

Proposition 9. *The accumulated complexity of a strategy bounds its complexity. Conversely, its accumulated complexity is polynomially bounded with respect to its complexity.*

Proof. If a strategy is computed (as a first-order function) by a Turing machine in time T_0, then the running time of the corresponding game machine defined in the proof of proposition 8 is the sum of the sizes of the opponent moves (*i.e.* oracle answers) plus the sum of the running times $T_0(p)$ for each odd prefix of the history: this is precisely the definition of the accumulated complexity.

Conversely, if s is computed by a game machine with complexity T, then for all strategies $s_1, \ldots, s_n$ such that $H(s, s_1, s_n) = m_1, m'_1, \ldots, m_t, m'_t$, the accumulated complexity T' of the Turing machine build from the game machine as described in the proof of proposition 8 verifies:

$$\forall 1 \leq k \leq t, T'(|m_1, m'_1, \ldots m_k|) \leq T(\|s_1\|_{\tau_1}, \ldots, \|s_n\|_{\tau_n})$$
$$|m_k| \leq T(\|s_1\|_{\tau_1}, \ldots, \|s_n\|_{\tau_n}).$$

By summing these inequalities, we obtain:

$$\sum_{1 \leq k \leq t} |m_k| + T'(|m_1, m'_1, \ldots m_k|) \leq 2t \cdot T(\|s_1\|_{\tau_1}, \ldots, \|s_n\|_{\tau_n}).$$

According to remark 9, the number $2t$ of moves in the history is itself bounded by $T(\|s_1\|_{\tau_1}, \ldots, \|s_n\|_{\tau_n})$. By maximizing over all argument strategies whose sizes are bounded by given bounds, we can conclude that the accumulated complexity is bounded quadratically with respect to T. □

The fact that the accumulated complexity bounds the complexity is not really surprising. Indeed, it can be seen as a rough upper bound on the computation time required to simulate a confrontation: it repeats unnecessary computations over increasing prefixes of the history whereas the game machine avoids them, by its interactive nature.

4.4. Polynomial time computable higher-order functions

Since size and complexity bounds of a strategy are functions of arbitrary finite type, we need a class of bounding functions for these types in order to generalize the usual classes of polynomial time computable functions for first and second-order functions. The usual polynomials have already been extended by Kapron and Cook [2] to define the class FPTIME_2 of polynomial time computable second-order functions. These polynomials have in fact also been extended to all finite types [8] in order to characterize BFF and define other higher-order complexity classes.

Definition 18 (Higher-order polynomials). Higher-order polynomials are terms of the simply-typed λ-calculus over base type $\mathbb{N}$ with constants for addition and multiplication of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

This class of functions is in particular stable by substitution, λ-abstraction and its restriction to first-order types (resp. second-order types) corresponds to the usual first-order (resp. second-order) polynomials.

We can now define the polynomial time computable strategies as strategies computed by a game machine whose running time is bounded by a higher-order polynomial. Note that according to proposition 9, using accumulated complexity instead would define the same class of strategies.

Definition 19. Let POLY_τ be the set of PCF functions of type τ represented by a (higher-order) polynomial time computable strategy.

In order to understand and justify this definition, we will need a few results to compare it with other existing classes. First, the polynomial PCF functions of type $\mathbb{N} \to \mathbb{N}$ and $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are the usual first and second-order polynomial time computable functions.

Proposition 10. *For all first-order or second-order type τ, $\text{BFF}_\tau = \text{POLY}_\tau$.*

Indeed, it is easy to see that the game machines for these two games respectively behave quite like usual Turing machines and oracle Turing machines. And since the usual notion of size for inputs of type $\mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$ coincide with our notion of size on the corresponding strategies (see propositions1 and 2), the notions of complexity are (polynomially) equivalent.

We will now prove a few lemmas to compare this class with the well known higher-order complexity class BFF.

Lemma 1. *For any finite type τ, the complexity of the identity function of type $\tau \to \tau$ is linear.*

Proof. The identity function is represented by the *copycat* strategy, which on each opponent move in one copy of the arena $\mathcal{A}_\tau$ repeats this move in the other copy of the arena $\mathcal{A}_\tau$. This strategy is innocent, since it depends only on the last played move.

If $\tau = \tau_1 \times \cdots \times \tau_n \to \beta$ and $s_1, \ldots, s_n$ are strategies in the corresponding subgames, then the size of the history of the copycat strategy against $s, s_1, \ldots s_n$ is roughly twice the size of the history of s against $s_1, \ldots, s_n$ since every move is repeated. For a more correct upper bound (three times is sufficient) we need to take into account the increase in the length of each justification name (at most one). In other words, its size is roughly bounded by $\lambda b.\lambda b_1 \ldots . \lambda b_n . 2 \cdot (b\ b_1\ \ldots\ b_n)$ which could also be written $\lambda b. 2 \cdot b$. This is also a bound on its complexity since the number of steps a game machine needs to copy an oracle answer as an oracle query is linear in the size of this query. □

Corollary 1. *The complexity of the projection functions of type $\tau \times \sigma \to \tau$ and $\tau \times \sigma \to \sigma$ is also linear.*

Indeed, they all have a strategy which are similar (if not equal to) the copycat strategy.

Corollary 2. *For any finite types σ and τ, the application function of type $(\sigma \to \tau) \times \sigma \to \tau$ is polynomial time computable.*

This is also a particular case of identity function. According to the proof of lemma 1, its complexity is bounded by $\lambda(x,y).2 \cdot (x\ y)$. In other words, the complexity of the application function is (roughly) the application function itself.

This implies that the class POLY is stable by composition. In particular, if F is computable in time P_1 and f in time P_2 (where P_1 and P_2 are higher-order polynomials), then the composition $F(f,\cdot)$ is computable in time $2 \cdot P_1(P_2,\cdot)$.

As it is expected of an analogue of FPTIME at higher types, this class extends BFF.

Theorem 3. *Every* BFF *function has a polynomial strategy (in the corresponding* PCF*-game).*

Proof. The BFF functions can be defined using the PV^ω language underlined in section 2. Proposition 10 covers the case of FPTIME, and the previous corollaries deal with the cases of application, composition, projections, and expansion.

Also, the second-order bounded recursion on notation operator can be seen as a second-order polynomial time computable function, which is covered by proposition 10. More precisely, we can prove that its complexity is roughly bounded by $\lambda n_0, G, Hn \mapsto n \cdot G(H(n, B(n) + n_0)) + n_0$ (it can be computed by $|x|$ iteration of F applied to x an input bounded by the size of B on x).

A BFF function is then polynomial time computable in our sense, and we can even build a bound on its complexity from a PV^ω term. □

However, from order 3 and above POLY strictly contains BFF. Indeed, the function Ψ from example 1 is computable in polynomial time.

Example 6. *A polynomial time computable strategy for Ψ can be described this way. It first simulates the confrontation (as the copycat strategy does) of its first argument (namely, a strategy for F) on a (simulated) strategy for f_x. Since f_x is computable in polynomial time with respect to $|x|$, say bounded by $\lambda y.P(|x|,y)$, this is done in time $G(\lambda y.P(n,y))$ if G and n respectively bound the sizes of the argument strategies (for F and x). Similarly, the second evaluation is done in time $G(\lambda y.c)$ where $\lambda y.c$ bounds the complexity of the zero function. Then the comparison of those values takes at most the same time. It remains to show that in the second case, the size of 2^x is bounded by a higher type polynomial in G and n. Indeed, if the strategy for F can distinguish between these two inputs, then in means it that during these two confrontations it has evaluated them at the only point where they differ, namely 2^x. In other words, the binary encoding of 2^x appears in the corresponding histories, so $|2^x|$ is bounded by $G(\lambda y.c)$ and $G(\lambda y.P(n,y))$. Overall, the complexity of this strategy is bounded by $\lambda G.\lambda n.G(\lambda y.P(n,y)) + G(\lambda y.c)$ up to a multiplicative constant.*

Overall, these results support the hypothesis that this new class naturally extends the usual class of polynomial time computable functions and does not suffer from the same weaknesses as BFF. Some additional results are required to show that it does not include any function which are not intuitively feasible. However, it is not yet clear how they should be stated, given the informal nature

of this notion. On the opposite, finding counter-examples, if they exist, would be simpler, and would help to better outline this notion of *higher-order feasibility*.

5. Conclusion and generalization to other games

In this paper, we focused only on defining complexity (and in particular polynomial time complexity) for higher-order functions, which was our main motivation but we now have a quite generic framework where we can define analogue notions like non-deterministic or space complexity since we use a quite generic machine model.

However, while doing so, we have underlined the essential elements which are necessary to define complexity in a more general setting. In particular, it is conceivable to extend our definitions of size and complexity for more general games as soon as they meet the following requirements:

- an explicit binary encoding of moves and names is provided (which forbids uncountable arenas);
- the arenas have finite depth and are acyclic, so that a notion of *argument game* can be defined (in which case, the order of the type associated to their size or complexity is still related to their depth);
- the plays verify the minimal rules (in particular: justified, well-opened, alternating and stable by sub-play) necessary to define the confrontation.

In particular, some elements of PCF-games are not necessary to define complexity, and additional rules can be allowed as soon as they are stable by sub-game or if another notion of sub-game is defined so that this is the case. For example, the innocence condition is only here to restrict the knowledge that the players have on the current play, which in particular guarantees their intentionality, *i.e.* that we can indeed interpret them as functions. But we could easily study games with strategies without this restriction, *i.e.* having access to the whole play and obtain a different notion of complexity for them, although we wouldn't be able to interpret them as higher-order functions anymore.

In addition, it seems a good start to define a notion of complexity for non-sequential games (*i.e.* where plays are not necessarily alternating), allowing for example to define the complexity of the *parallel or* function, but this is a more difficult task.

It is now necessary to provide more arguments in favor of this class and this notion of complexity in general. In particular, it would be interesting to have a characterization using a function algebra, similar to Cobham's for FPTIME, Mehlhorn's for FPTIME_2, or PV^ω for BFF. Also, for a functional languages approach, a connection with notions like *derivational complexity* [23, 24, 25, 26] could be made.

Acknowledgments

I would like to thank the anonymous reviewers for their very relevant remarks which greatly helped improve the general clarity of this paper, as well as those who supported me while writing it, amongst whom were Mathieu Hoyrup and Martin Ziegler.

References

[1] K. Mehlhorn, Polynomial and abstract subrecursive classes, J. Comput. Syst. Sci. 12 (2) (1976) 147–178.

[2] B. M. Kapron, S. A. Cook, A new characterization of type-2 feasibility, SIAM Journal on Computing 25 (1) (1996) 117–132.

[3] A. Kawamura, S. Cook, Complexity theory for operators in analysis, in: Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC '10, ACM, New York, NY, USA, 2010, pp. 495–502.

[4] H. Férée, M. Hoyrup, W. Gomaa, On the query complexity of real functionals, in: LICS - 28th ACM/IEEE Symposium on Logic in Computer Science, New Orleans, États-Unis, 2013, pp. 103–112.

[5] H. Férée, W. Gomaa, M. Hoyrup, Analytical properties of resource-bounded real functionals, Journal of Complexity 30 (5) (2014) 647–671.

[6] S. A. Cook, B. M. Kapron, Characterizations of the basic feasible functionals of finite type, Foundations of Computer Science, IEEE Annual Symposium on 0 (1989) 154–159.

[7] A. Seth, Turing machine characterizations of feasible functionals of all finite types, Feasible Mathematics II (1995) 407–428.

[8] R. J. Irwin, J. S. Royer, B. M. Kapron, On characterizations of the basic feasible functionals (part ii), unpublished (2002).

[9] R. J. Irwin, J. S. Royer, B. M. Kapron, On characterizations of the basic feasible functionals (part i), J. Funct. Program. 11 (1) (2001) 117–153.

[10] S. Bellantoni, Comments on two notions of higher type computability., Unpublished notes.

[11] S. C. Kleene, Countable functionals, Constructivity in Mathematics (1959) 81–100.

[12] H. Nickau, D. D. Naturwissenschaften, D. m. H. Nickau, A. V. Fachbereich, D. Prof, D. F.-j. Delvos, Hereditarily sequential functionals: A game-theoretic approach to sequentiality (1996).

[13] J. M. E. Hyland, C. H. L. Ong, On full abstraction for pcf: I, ii, and iii, Inf. Comput. 163 (2) (2000) 285–408.

[14] G. D. Plotkin, Lcf considered as a programming language, Theoretical Computer Science 5 (3) (1977) 223–255.

[15] A. Cobham, The intrinsic computational difficulty of functions (1965) 24.

[16] S. Cook, A. Urquhart, Functional interpretations of feasibly constructive arithmetic, Annals of Pure and Applied Logic 63 (2) (1993) 103–200.

[17] S. R. Buss, B. M. Kapron, Resource-bounded continuity and sequentiality for type-two functionals, ACM Trans. Comput. Logic 3 (3) (2002) 402–417.

[18] J. S. Royer, On the computational complexity of longley's h functional, in: Second International Workshop on Implicit Computational Complexity, UC/Santa Barbara, 2000.

[19] J. R. Longley, Notions of computability at higher types i, in: Logic Colloquium, Vol. 19, 2000, pp. 32–142.

[20] M. J. Gabbay, D. R. Ghica, Game semantics in the nominal model (2012) 173–189

[21] M. J. Gabbay, V. Ciancia, Freshness and name-restriction in sets of traces with names, in: Foundations of software science and computation structures, 14th International Conference (FOSSACS 2011), Vol. 6604 of Lecture Notes in Computer Science, Springer, 2011, pp. 365–380.

[22] H. Nickau, Lecture notes in computer science, in: A. Nerode, Y. V. Matiyasevich (Eds.), Logical Foundations of Computer Science, Vol. 813, Springer Berlin Heidelberg, 1994, Ch. Hereditarily sequential functionals, pp. 253–264.

[23] D. Sands, Complexity analysis for a lazy higher-order language, in: N. D. Jones (Ed.), ESOP, Vol. 432 of Lecture Notes in Computer Science, Springer, 1990, pp. 361–376.

[24] N. Danner, J. Paykin, J. S. Royer, A static cost analysis for a higher-order language, in: M. Might, D. V. Horn, A. Abel, T. Sheard (Eds.), PLPV, Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013, ACM, 2013, pp. 25–34.

[25] N. Danner, J. S. Royer, Adventures in time and space, Logical Methods in Computer Science 3 (1).

[26] U. Dal Lago, S. Martini, Lecture notes in computer science, in: M. van Eekelen, O. Shkaravska (Eds.), Foundational and Practical Aspects of Resource Analysis, Vol. 6324, Springer Berlin Heidelberg, 2010, Ch. Derivational Complexity Is an Invariant Cost Model, pp. 100–113.