Characterizing polynomial time complexity of stream programs using interpretations

Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux

Characterizing polynomial time complexity of stream programs using interpretations

Hugo Férée[a,c], Emmanuel Hainry[a,c], Mathieu Hoyrup[b,c], Romain Péchoux[a,c]

[a] *Université de Lorraine, Nancy, France*
[b] *Inria Nancy - Grand Est, Villers-lès-Nancy, France*
[c] *Project-team CARTE, LORIA, UMR7503*

Abstract

This paper provides a criterion based on interpretation methods on term rewrite systems in order to characterize the polynomial time complexity of second order functionals. For that purpose it introduces a first order functional stream language that allows the programmer to implement second order functionals. This characterization is extended through the use of exp-poly interpretations as an attempt to capture the class of Basic Feasible Functionals (BFF). Moreover, these results are adapted to provide a new characterization of polynomial time complexity in computable analysis. These characterizations give a new insight on the relations between the complexity of functional stream programs and the classes of functions computed by Oracle Turing Machine, where oracles are treated as inputs.

Keywords:
Stream Programs, Type-2 Functionals, Interpretations, Polynomial Time, Basic Feasible Functionals, Computable Analysis, Rewriting

1. Introduction

Lazy functional languages like Haskell allow the programmer to deal with co-inductive datatypes in such a way that co-inductive objects can be evaluated by finitary means. Consequently, computations over streams, that is infinite lists, can be performed in such languages.

A natural question arising is the complexity of the programs computing on streams. Intuitively, the complexity of a stream program is the number of reduction steps needed to output the n first elements of a stream, for any n. However the main issue is to relate the complexity bound to the input structure. Since a stream can be easily identified with a function, a good way for solving such an issue is to consider computational and complexity models dealing with functions as inputs.

In this perspective, we want to take advantage of the complexity results obtained on type-2 functions (functions over functions), and in particular on BFF [1, 2], to understand stream program complexity. For that purpose, we

set Unary Oracle Turing Machine (UOTM), machines computing functions with oracles taking unary inputs, as our main computational model. This model is well-suited in our framework since it manipulates functions as objects with a well-defined notion of complexity. UOTM are a refinement of Oracle Turing Machines (OTM) on binary words which correspond exactly to the BFF algebra in [3] under polynomial restrictions. UOTM are better suited than OTM to study stream complexity with a realistic complexity measure, since in the UOTM model accessing the n^{th} element costs n transitions whereas it costs $\log(n)$ in the OTM model.

The Implicit Computational Complexity (ICC) community has proposed characterizations of OTM complexity classes using function algebra [3, 4] and type systems [5, 6] or recently as a logic [7].

These latter characterizations are inspired by former characterizations of type-1 polynomial time complexity based on ramification [8, 9]. This line of research has led to new developments of other ICC tools and in particular to the use of (polynomial) interpretations in order to characterize the classes of functions computable in polynomial time or space [10, 11].

Polynomial interpretations [12, 13] are a well-known tool used to show the termination of first order term rewrite systems. This tool has been adapted into variants, like quasi-interpretations and sup-interpretation [14], that allow the programmer to analyze program complexity. In general, interpretations are restricted to inductive data types and [15] was a first attempt to adapt such a tool to co-inductive data types including stream programs. In this paper, we introduce a second order variation of this interpretation methodology in order to constrain the complexity of stream program computation and we obtain a characterization of UOTM polynomial time computable functions. Using this characterization, we can analyze functions of this class in an easier way based on the premise that it is practically easier to write a first order functional program on streams than the corresponding Unary Oracle Turing Machine. The drawback is that the tool suffers from the same problems as polynomial interpretation: the difficulty to automatically synthesize the interpretation of a given program (see [16]). As a proof of versatility of this tool, we provide a partial characterization of the BFF class (the full characterization remaining open), just by changing the interpretation codomain: for that purpose, we use restricted exponentials instead of polynomials in the interpretation of a stream argument.

A direct and important application is that second order polynomial interpretations on stream programs can be adapted to characterize the complexity of functions computing over reals defined in Computable Analysis [17]. This approach is a first attempt to study the complexity of such functions through static analysis methods.

This paper is an extended version of [18] with complete proofs, additional examples and corrections.

Outline of the paper.

In Section 2, we introduce (Unary) Oracle Turing Machines and their complexity. In Section 3, we introduce the studied first order stream language. In

Section 4, we define the interpretation tools extended to second order and we provide a criterion on stream programs. We show our main characterization relying on the criterion in Section 5. Section 6 develops a new application, which was only mentioned in [18], to functions computing over reals.

2. Polynomial time Oracle Turing Machines

In this section, we will define a machine model and a notion of complexity relevant for stream computations. This model (UOTM) is adapted from the Oracle Turing Machine model used by Kapron and Cook in their characterization of Basic Feasible functionals (BFF) [3]. In the following, $|x|$ will denote the size of the binary encoding of $x \in \mathbb{N}$, namely $\lceil \log_2(x) \rceil$.

Definition 1 (Oracle Turing Machine). An Oracle Turing Machine (denoted by OTM) $\mathcal{M}$ with k oracles and l input tapes is a Turing machine with, for each oracle, a state, one query tape and one answer tape.

Whenever $\mathcal{M}$ is used with input oracles $F_1, \dots F_k : \mathbb{N} \longrightarrow \mathbb{N}$ and arrives on the oracle state $i \in \{1, \dots, k\}$ and if the corresponding query tape contains the binary encoding of a number x, then the binary encoding of $F_i(x)$ is written on the corresponding answer tape. It behaves like a standard Turing machine on the other states.

We now introduce the unary variant of this model, which is more related to stream computations as accessing the n-th element takes at least n steps (whereas it takes $\log(n)$ steps in OTM. See example 2 for details).

Definition 2 (Unary Oracle Turing Machine). A Unary Oracle Turing Machine (denoted UOTM) is an OTM where numbers are written using unary notation on the query tape, *i.e.* on the oracle state i, if w is the content of the corresponding query tape, then $F_i(|w|)$ is written on the corresponding answer tape.

Definition 3 (Running time). In both cases (OTM and UOTM), we define the cost of a transition as the size of the answer of the oracle, in the case of a query, and 1 otherwise.

In order to introduce a notion of complexity, we have to define the size of the inputs of our machines.

Definition 4 (Size of a function). The size $|F| : \mathbb{N} \longrightarrow \mathbb{N}$ of a function $F : \mathbb{N} \longrightarrow \mathbb{N}$ is defined by:

$$|F|(n) = \max_{k \leq n} |F(k)|$$

Remark 1. This definition is different from the one used in [3] (denoted here by $||.||$). Indeed, the size of a function was defined by $||F||(n) = \max_{|k| \leq n} |F(k)|$, in other words, $||f||(n) = |f|(2^n - 1)$. The reason for this variation is that in an UOTM, the oracle is closer to an infinite sequence (or stream as we will see in the

following) than to a function, since it can access easily its first n elements but not its $(2^n)^{th}$ element which is the case in an OTM. In particular, this makes the size function computable in polynomial time (with respect to the following definitions).

The size of an oracle input is then a type-1 function whereas it is an integer for standard input. Then, the notion of polynomial running time needs to be adapted.

Definition 5 (Second order polynomial). A second order polynomial is a polynomial generated by the following grammar:

$$P := c \mid X \mid P + P \mid P \times P \mid Y\langle P\rangle$$

where X represents a zero order variable (ranging over $\mathbb{N}$), Y a first order variable (ranging over $\mathbb{N} \longrightarrow \mathbb{N}$), c a constant in $\mathbb{N}$ and $\langle -\rangle$ is a notation for the functional application.

Example 1. $P(Y_1, Y_2, X_1, X_2) = (Y_1\langle Y_1\langle X_1\rangle \times Y_2\langle X_2\rangle\rangle)^2$ is a second order polynomial.

The following lemma shows that the composition of polynomials is well behaved.

Lemma 1. *If the first order variables of a second order polynomial are replaced with first order polynomials, then the resulting function is a first order polynomial.*

PROOF. This can be proved by induction on the definition of a second order polynomial: this is true if P is constant or equal to a zero order variable and also if P is a sum, a multiplication or a composition with type-1 variable, since polynomials are stable under these operators.

In the following, $P(Y_1, \ldots, Y_k, X_1, \ldots, X_l)$ will denote a second order polynomial where each Y_i represents a first order variable, and each X_i a zero order variable.

This definition of running time is directly adapted from the definition of running time for OTMs.

Definition 6 (Polynomial running time). An UOTM $\mathcal{M}$ operates in time $T : (\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow \mathbb{N}$ if for all inputs $x_1, \ldots x_l : \mathbb{N}$ and $F_1, \ldots F_k : \mathbb{N} \longrightarrow \mathbb{N}$, the sum of the transition costs before $\mathcal{M}$ halts on these inputs is less than $T(|F_1|, \ldots, |F_k|, |x_1|, \ldots, |x_l|)$.

A function $G : (\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow \mathbb{N}$ is UOTM computable in polynomial time if there exists a second order polynomial P such that G is computed by an UOTM in time P.

The class BFF is defined in a similar manner substituting $||.||$ to $|.|$.

Lemma 2. *The set of polynomial time* UOTM *computable functions is strictly included in the set of polynomial time* OTM *computable functions (proved to be equal to the* BFF *algebra [3]):*

PROOF. In order to transform a UOTM into an OTM computing the same functional, we have to convert the content of the query tape from the word w (representing $|w|$ in unary) into the binary encoding of $|w|$ before each oracle call. This can be done in polynomial time in $|w|$ and we call Q this polynomial. In both cases, the cost of the transition is $|F(|w|)|$, so the running time of both machines is the same, except for the conversion time. If the computation time of the UOTM was bounded by a second order polynomial P, the size of the content of the query is at most $P(|F|)$ at each query, so the additional conversion time is at most $Q(P(|F|))$ for each query and there are at most $P(|F|)$ queries. Thus, the conversion time is also a second order polynomial in the size of the inputs, and the computation time of the OTM is also bounded by a second order polynomial in $|F|$. Finally, since $||F||$ bounds $|F|$, the computation time is also bounded by the same polynomial in $||F||$, so F is in BFF.

Example 2. The function $G : (\mathbb{N} \longrightarrow \mathbb{N}) \times \mathbb{N} \longrightarrow \mathbb{N}$ defined by $G(F, x) = F(|x|) = F(\lceil \log_2(x) \rceil)$ is UOTM computable in polynomial time. Indeed, it is computed by the function which copies the input word representing x on the query tape before entering the query state before returning the content of the answer tape as output. Its running time is bounded by $2 \times (|x| + |F|(|x|))$.

However $H(F, x) = F(x)$ is not UOTM computable in polynomial time, because it would require to write x in unary on the query tape, which costs $2^{|x|}$. Nonetheless, H is in BFF because an OTM only has to write x in binary, and the oracle call costs $|F(x)| \leq ||F||(|x|)$.

3. First order stream language

3.1. Syntax

In this section, we define a simple Haskell-like lazy first order functional language with streams.

$\mathcal{F}$ will denote the set of function symbols, $\mathcal{C}$ the set of constructor symbols and $\mathcal{X}$ the set of variable names. A program is a list of definitions $\mathcal{D}$ given by the grammar in Figure 1:

$$\begin{aligned}
\mathtt{p} &::= \mathtt{x} \mid \mathtt{c}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n \mid \mathtt{p} : \mathtt{y} && \text{(Patterns)}\\
\mathtt{e} &::= \mathtt{x} \mid \mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n && \text{(Expressions)}\\
\mathtt{d} &::= \mathtt{f}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n = \mathtt{e} && \text{(Definitions)}
\end{aligned}$$

Figure 1: Program grammar

where $\mathtt{x}, \mathtt{y} \in \mathcal{X}, \mathtt{t} \in \mathcal{C} \cup \mathcal{F}, \mathtt{c} \in \mathcal{C} \setminus \{:\}$ and $\mathtt{f} \in \mathcal{F}$ and $\mathtt{c}, \mathtt{t}$ and $\mathtt{f}$ are symbols of arity n.

Throughout the paper, we call closed expression any expression without variables.

The stream constructor $: \in \mathcal{C}$ is a special infix constructor of arity 2. In a stream expression $\mathtt{h} : \mathtt{t}$, $\mathtt{h}$ and $\mathtt{t}$ are respectively called the head and the tail of the stream.

We might use other infix or postfix constructors or function symbols in the following for ease of readability.

In a definition $\mathtt{f}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n = \mathtt{e}$, all the variables of $\mathtt{e}$ appear in the patterns $\mathtt{p}_i$. Moreover patterns are non overlapping and each variable appears at most once in the left-hand side. This entails that programs are confluent. In a program, we suppose that all pattern matchings are exhaustive. Finally, we only allow patterns of depth 1 for the stream constructor (*i.e.* only variables appear in the tail of a stream pattern).

Remark 2. This is not restrictive since a program with higher pattern matching depth can be easily transformed into a program of this form using extra function symbols and definitions. We will prove in Proposition 1 that this modification does not alter our results on polynomial interpretations.

3.2. Type system

Programs contain inductive types that will be denoted by `Tau`. For example, unary integers are defined by:

```
data Nat = 0 | Nat +1
```

(with $\mathtt{0}, \mathtt{+1} \in \mathcal{C}$), and binary words by:

```
data Bin = Nil | 0 Bin | 1 Bin.
```

Consequently, each constructor symbol comes with a typed signature and we will use the notation `c :: T` to denote that the constructor symbol `c` has type `T`. For example, we have `0 :: Nat` and `+1 :: Nat -> Nat`. Note that in the following, given some constants $n, k, \ldots \in \mathbb{N}$, the terms $\mathtt{n}$, $\mathtt{k}$,... denote their encoding as unary integers in `Nat`.

Programs contain co-inductive types defined by `data [Tau] = Tau : [Tau]` for each inductive type `Tau`. This is a distinction with Haskell, where streams are defined to be both finite and infinite lists, but not a restriction since finite lists may be defined in this language and since we are only interested in showing properties of total functions (*i.e.* an infinite stream represents a total function).

In the following, we will write $\mathtt{T}^k$ for `T -> T -> ...-> T` (with k occurrences of `T`).

Each function symbol `f` comes with a typed signature that we restrict to be either $\mathtt{f::[Tau]}^k$ `-> Tau`^{l} `-> Tau` or $\mathtt{f::[Tau]}^k$ `-> Tau`^{l} `-> [Tau]`, with $k, l \geq 0$.

Throughout the paper, we will only consider well-typed programs where the left-hand side and the right-hand side of a definition can be given the same type using the simple rules of Figure 2 with $\mathtt{A}, \mathtt{A}_i \in \{\mathtt{Tau}, \mathtt{[Tau]}\}$. Γ is a typing basis for every variable, constructor and function symbol (we assume that each variable is used in at most one definition, for the sake of simplicity).

$$\frac{\Gamma(\mathtt{x}) = \mathtt{A}}{\Gamma \vdash \mathtt{x} :: \mathtt{A}}\ \mathtt{x} \in \mathcal{X} \cup \mathcal{C} \cup \mathcal{F}$$

$$\frac{\forall(\mathtt{f}\ \mathtt{p}_1 \ldots \mathtt{p}_n = \mathtt{e}) \in \mathcal{D}, \qquad \Gamma \vdash \mathtt{e} :: \mathtt{A} \qquad \forall i, \Gamma \vdash \mathtt{p}_i :: \mathtt{A}_i}{\Gamma \vdash \mathtt{f} :: \mathtt{A}_1\ \mathtt{->}\ \ldots\ \mathtt{->}\ \mathtt{A}_n\ \mathtt{->}\ \mathtt{A}}\ \mathtt{f} \in \mathcal{F}$$

$$\frac{\Gamma \vdash \mathtt{t} :: \mathtt{A}_1\ \mathtt{->}\ \ldots\ \mathtt{->}\ \mathtt{A}_n\ \mathtt{->}\ \mathtt{A} \qquad \forall i \in \{1, \ldots, n\},\ \mathtt{e}_i :: \mathtt{A}_i}{\Gamma \vdash \mathtt{t}\ \mathtt{e}_1 \ldots \mathtt{e}_n :: \mathtt{A}}\ \mathtt{t} \in \mathcal{C} \cup \mathcal{F}$$

Figure 2: Typing rules

3.3. Semantics

Lazy values and strict values are defined in Figure 3:

$$\begin{aligned} \mathtt{lv} &::= \mathtt{e}_1 : \mathtt{e}_2 && \text{(Lazy value)} \\ \mathtt{v} &::= \mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_n && \text{(Strict value)} \end{aligned}$$

Figure 3: Values and lazy values

where $\mathtt{e}_1, \mathtt{e}_2$ are closed expressions and $\mathtt{c}$ belongs to $\mathcal{C} \setminus \{:\}$. Lazy values are stream expressions with the constructor symbol : at the top level whereas strict values are expressions of inductive type where only constructor symbols occur and are used to deal with fully evaluated elements.

Moreover, let $\mathfrak{S}$ represent the set of substitutions σ that map variables to expressions. As usual the result of applying the substitution σ to an expression $\mathtt{e}$ is denoted $\sigma(\mathtt{e})$.

The evaluation rules are defined in Figure 4:

$$\frac{(\mathtt{f}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n = \mathtt{e}) \in \mathcal{D} \qquad \sigma \in \mathfrak{S} \qquad \forall i \in \{1,...,n\},\ \sigma(\mathtt{p}_i) = \mathtt{e}_i}{\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n \to \sigma(\mathtt{e})}\ (d)$$

$$\frac{\mathtt{e}_i \to \mathtt{e}_i' \qquad \mathtt{t} \in \mathcal{F} \cup \mathcal{C} \setminus \{:\}}{\mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i\ \ldots\ \mathtt{e}_n \to \mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i'\ \ldots\ \mathtt{e}_n}\ (t)$$

$$\frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{e} : \mathtt{e}_0 \to \mathtt{e}' : \mathtt{e}_0}\ (:)$$

Figure 4: Evaluation rules

We will write $\mathtt{e} \to^n \mathtt{e}'$ if there exist expressions $\mathtt{e}_1, \ldots, \mathtt{e}_{n-1}$ such that $\mathtt{e} \to \mathtt{e}_1 \cdots \to \mathtt{e}_{n-1} \to \mathtt{e}'$. Let $\to^*$ denote the transitive and reflexive closure of $\to$. We write $\mathtt{e} \to_! \mathtt{e}'$ if $\mathtt{e}$ is normalizing to the expression $\mathtt{e}'$, *i.e.* $\mathtt{e} \to^* \mathtt{e}'$ and there is no $\mathtt{e}''$ such that $\mathtt{e}' \to \mathtt{e}''$. We can show easily wrt the evaluation rules (and because definitions are exhaustive) that given a closed expression $\mathtt{e}$, if $\mathtt{e} \to_! \mathtt{e}'$ and $\mathtt{e} :: \mathtt{Tau}$ then $\mathtt{e}'$ is a strict value, whereas if $\mathtt{e} \to_! \mathtt{e}'$ and $\mathtt{e} :: [\mathtt{Tau}]$ then $\mathtt{e}'$ is a lazy value. Indeed the (t) rule of Figure 4 allows the reduction of an expression under a function or constructor symbol whereas the (:) rule only allows reduction of a stream head (this is why stream patterns of depth greater than 1 are not allowed in a definition).

These reduction rules are not deterministic but a call-by-need strategy could be defined to mimic Haskell's semantic.

The following function symbols are typical stream operators and will be used further.

Example 3. `s !! n` computes the $(n+1)^{th}$ element of the stream `s`:

```
!! :: [Tau] -> Nat -> Tau
(h:t) !! (n+1) = t !! n
(h:t) !! 0 = h
```

Example 4. `tln s n` drops the first $n+1$ elements of the stream `s`:

```
tln :: [Tau] -> Nat -> [Tau]
tln (h:t) (n+1) = tln t n
tln (h:t) 0 = t
```

4. Second order polynomial interpretations

In the following, we will call a **positive functional** any function of type $(\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow T$ with $k, l \in \mathbb{N}$ and $T \in \{\mathbb{N}, \mathbb{N} \longrightarrow \mathbb{N}\}$.

Given a positive functional $F : ((\mathbb{N} \longrightarrow \mathbb{N})^k \times \mathbb{N}^l) \longrightarrow T$, the arity of F is $k + l$.

Let $>$ denote the usual ordering on $\mathbb{N}$ and its standard extension to $\mathbb{N} \longrightarrow \mathbb{N}$, *i.e.* given $F, G : \mathbb{N} \longrightarrow \mathbb{N}$, $F > G$ if $\forall n \in \mathbb{N} \setminus \{0\}, F(n) > G(n)$ (the comparison on 0 is not necessary since we will only use strictly positive inputs).

We extend this ordering to the set of positive functionals of arity $k + l$ by: $F > G$ if

$$\forall y_1, \dots y_k \in \mathbb{N} \longrightarrow^{\uparrow} \mathbb{N}, \forall x_1 \dots x_l \in \mathbb{N} \setminus \{0\},$$

$$F(y_1, \dots y_k, x_1, \dots, x_l) > G(y_1, \dots y_k, x_1, \dots, x_l)$$

where $\mathbb{N} \longrightarrow^{\uparrow} \mathbb{N}$ is the set of increasing functions on positive integers.

Definition 7 (Monotonic positive functionals). A positive functional is monotonic if it is strictly increasing with respect to each of its arguments.

Definition 8 (Types interpretation). The type signatures of a program are interpreted as types this way:

- an inductive type `Tau` is interpreted as $\mathbb{N}$
- a stream type `[Tau]` is interpreted as $\mathbb{N} \longrightarrow \mathbb{N}$
- an arrow type `A -> B` is interpreted as the type $T_{\mathtt{A}} \longrightarrow T_{\mathtt{B}}$ if `A` and `B` are respectively interpreted as $T_{\mathtt{A}}$ and $T_{\mathtt{B}}$.

Definition 9 (Assignment). An assignment is a mapping of each function symbol `f` :: `A` to a monotonic positive functional whose type is the interpretation of `A`.

An assignment can be canonically extended to any expression in the program:

Definition 10 (Expression assignment). The assignment is defined on constructors and expressions this way:

- $(\!|\mathtt{c}|\!)(X_1, \dots, X_n) = \sum_{i=1}^{n} X_i + 1$, if $\mathtt{c} \in \mathcal{C} \setminus \{:\}$ is of arity n.
- $\begin{cases} (\!|:|\!)(X, Y)(0) = X \\ (\!|:|\!)(X, Y)(Z + 1) = 1 + X + Y\langle Z \rangle \end{cases}$
- $(\!|\mathtt{x}|\!) = X$, if `x` is a variable of type `Tau`, *i.e.* we associate a unique zero order variable X in $\mathbb{N}$ to each $\mathtt{x} \in \mathcal{X}$ of type `Tau`.
- $(\!|\mathtt{y}|\!)(Z) = Y\langle Z \rangle$, if `y` is a variable of type [`Tau`], *i.e.* we associate a unique first order variable $Y : \mathbb{N} \longrightarrow \mathbb{N}$ to each $\mathtt{y} \in \mathcal{X}$ of type [`Tau`].
- $(\!|\mathtt{t\ e}_1\ \dots\ \mathtt{e}_n|\!) = (\!|\mathtt{t}|\!)((\!|\mathtt{e}_1|\!), \dots, (\!|\mathtt{e}_n|\!))$, if $\mathtt{t} \in \mathcal{C} \cup \mathcal{F}$.

Remark 3. For every expression $\mathtt{e}$, $(\!|\mathtt{e}|\!)$ is a monotonic positive functional, since it is true for function symbols, for constructors, and the composition of such functionals is still monotonic positive.

Example 5. The stream constructor : has type `Tau -> [Tau] -> [Tau]`. Consequently, its assignment $(\!|:|\!)$ has type $(\mathbb{N} \times (\mathbb{N} \longrightarrow \mathbb{N})) \longrightarrow (\mathbb{N} \longrightarrow \mathbb{N})$. Let us consider the expression $\mathtt{p} : (\mathtt{q} : \mathtt{r})$, with $\mathtt{p}, \mathtt{q}, \mathtt{r} \in \mathcal{X}$, we obtain that:

$$\begin{aligned}(\!|\mathtt{p} : (\mathtt{q} : \mathtt{r})|\!) &= (\!|:|\!)((\!|\mathtt{p}|\!), (\!|\mathtt{q} : \mathtt{r}|\!)) = (\!|:|\!)((\!|\mathtt{p}|\!), (\!|:|\!)((\!|\mathtt{q}|\!), (\!|\mathtt{r}|\!))) = (\!|:|\!)(P, (\!|:|\!)(Q, R)) \\ &= F(R, Q, P)\end{aligned}$$

where $F \in ((\mathbb{N} \longrightarrow \mathbb{N}) \times \mathbb{N}^2) \longrightarrow (\mathbb{N} \longrightarrow \mathbb{N})$ is the positive functional such that:

- $F(R, Q, P)(Z + 2) = 1 + P + (\!|:|\!)(Q, R)(Z + 1) = 2 + P + Q + R(Z)$
- $F(R, Q, P)(1) = 1 + P + (\!|:|\!)(Q, R)(0) = 1 + P + Q$
- $F(R, Q, P)(0) = P$

Lemma 3. *The assignment of an expression* $\mathtt{e}$ *defines a positive functional in the assignment of its free variables (and an additional type-*0 *variable if* $\mathtt{e} :: [\mathtt{Tau}]$*).*

Proof. By structural induction on expressions. This is the case for variables and for the stream constructor, the inductive constructors are additive, and the assignments of function symbols are positive (the composition of a positive functional with positive functional being a positive functional).

Definition 11 (Polynomial interpretation). An assignment $(\!|-|\!)$ of the function symbols of a program defines an interpretation if for each definition $\mathtt{f}\ \mathtt{p}_1 \ldots \mathtt{p}_n = \mathtt{e} \in \mathcal{D}$,

$$(\!|\mathtt{f}\ \mathtt{p}_1 \ldots \mathtt{p}_n|\!) > (\!|\mathtt{e}|\!).$$

Furthermore, $(\!|\mathtt{f}|\!)$ is polynomial if it is bounded by a second order polynomial. In this case, the program is said to be polynomial.

The following programs will be used further and have polynomial interpretations:

Example 6. The sum over unary integers:

```
plus :: Nat -> Nat -> Nat
plus 0 b = b
plus (a+1) b = (plus a b)+1
```

`plus` admits the following (polynomial) interpretation:

$$(\!|\mathtt{plus}|\!)(X_1, X_2) = 2 \times X_1 + X_2$$

Indeed, we check that the following inequalities are satisfied:

$(\!|\mathtt{plus\ 0\ b}|\!) = 2 + B > B = (\!|\mathtt{b}|\!)$

$(\!|\mathtt{plus\ (a+1)\ b}|\!) = 2A + 2 + B > 2A + B + 1 = (\!|\mathtt{(plus\ a\ b)+1}|\!)$

Example 7. The product of unary integers:

```
mult :: Nat -> Nat -> Nat
mult 0 b = 0
mult (a+1) b = plus b (mult a b)
```

The following inequalities show that $(\!|\texttt{mult}|\!)(X_1, X_2) = 3 \times X_1 \times X_2$ is an interpretation for `mult`:

$$\begin{aligned}(\!|\texttt{mult 0 b}|\!) &= 3 \times (\!|\texttt{0}|\!) \times (\!|\texttt{b}|\!) = 3 \times B > 1 = (\!|\texttt{0}|\!)\\ (\!|\texttt{mult (a+1) b}|\!) &= 3 \times A \times B + 3 \times B\\ &> 2 \times B + 3 \times A \times B = (\!|\texttt{plus b (mult a b)}|\!)\end{aligned}$$

Note that these interpretations are first order polynomials because `plus` and `mult` only have inductive arguments.

Example 8. The function symbol `!!` defined in Example 3 admits an interpretation of type $((\mathbb{N} \longrightarrow \mathbb{N}) \times \mathbb{N}) \longrightarrow \mathbb{N}$ defined by:

$$(\!|\texttt{!!}|\!)(Y, N) = Y\langle N\rangle.$$

Indeed, we check that:

$$(\!|\texttt{(h:t) !! (n+1)}|\!) = (\!|\texttt{h:t}|\!)((\!|\texttt{n}|\!) + 1) = 1 + (\!|\texttt{h}|\!) + (\!|\texttt{t}|\!)((\!|\texttt{n}|\!)) > (\!|\texttt{t}|\!)((\!|\texttt{n}|\!)) = (\!|\texttt{t !! n}|\!)$$
$$(\!|\texttt{(h:t) !! 0}|\!) = (\!|\texttt{h:t}|\!)((\!|\texttt{0}|\!)) = (\!|\texttt{h:t}|\!)(1) = 1 + (\!|\texttt{h}|\!) + (\!|\texttt{t}|\!)(0) > (\!|\texttt{h}|\!)$$

Example 9. The function symbol `tln` defined in Example 4 admits an interpretation of type $((\mathbb{N} \longrightarrow \mathbb{N}) \times \mathbb{N}) \longrightarrow (\mathbb{N} \longrightarrow \mathbb{N})$ defined by:

$$(\!|\texttt{tln}|\!)(Y, N)(Z) = Y\langle N + Z + 1\rangle.$$

Indeed:

$$(\!|\texttt{tln (h:t) (n+1)}|\!)(Z) = 1 + H + T(N + Z) > T(N + Z) = (\!|\texttt{tln t n}|\!)(Z)$$

Theorem 1. *The synthesis problem, i.e. does a program admit a polynomial interpretation, is undecidable.*

Proof. It has already been proved in [19] that the synthesis problem is undecidable for usual integer polynomials and such term rewriting systems without streams. Since our programs include them, and that their interpretations are necessarily first order integer polynomials, the synthesis problem for our stream programs is also undecidable.

Now that we have polynomial interpretations for `!!` and `tln`, we can prove that restricting to stream patterns of depth 1 was not a loss of generality.

Proposition 1. *If a program with arbitrary depth on stream patterns has a polynomial interpretation, then there exists an equivalent depth-*1 *program with a polynomial interpretation.*

PROOF. If $\mathtt{f}$ has one stream argument using depth $k+1$ pattern matching in a definition of the shape $\mathtt{f}(\mathtt{p}_1 : (\mathtt{p}_2 : \ldots (\mathtt{p}_{k+1} : \mathtt{t}) \ldots)) = \mathtt{e}$, we transform $\mathtt{f}$ in a new equivalent program using `!!` and `tln` defined in Examples 3 and 4, and an auxiliary function symbol $\mathtt{f}_1$

```
f s =  f1 (tln s k) (s !! 0) ... (s !! k)
f1 t  p1 ...  pk+1 = e
```

This new definition of $\mathtt{f}$ is equivalent to the initial one, and if it admitted a polynomial interpretation P, then the equivalent function symbol $\mathtt{f}_1$ can be interpreted by:

$$(\!|\mathtt{f}_1|\!)(T, X_1, \ldots, X_{k+1})(Z) = P(\sum_{1 \leq i \leq k+1} X_i + T\langle Z \rangle + k + 1).$$

Then, we redefine $(\!|\mathtt{f}|\!)$ by:

$$(\!|\mathtt{f}|\!)(S)(Z) = 1 + (\!|\mathtt{f}_1|\!)(S, S\langle 1 \rangle, S\langle 2 \rangle, \ldots S\langle k+1 \rangle)(Z)$$

which is greater than:

$$(\!|\mathtt{f}_1|\!)((\!|\mathtt{s}|\!), (\!|\mathtt{s\ !!\ 0}|\!), \ldots, (\!|\mathtt{s\ !!\ (k)}|\!))(Z)$$

so it is indeed a polynomial interpretation for this modified $\mathtt{f}$.

Lemma 4. *If* $\mathtt{e}$ *is an expression of a program with an interpretation* $(\!|-|\!)$ *and* $\mathtt{e} \to \mathtt{e}'$*, then* $(\!|\mathtt{e}|\!) > (\!|\mathtt{e}'|\!)$.

PROOF. If $\mathtt{e} \to \mathtt{e}'$ using:

- the (d) rule, then there are a substitution σ and a definition $\mathtt{f\ p}_1 \ldots \mathtt{p}_n = \mathtt{d}$ such that $\mathtt{e} = \sigma(\mathtt{f\ p}_1 \ldots\ \mathtt{p}_n)$ and $\mathtt{e}' = \sigma(\mathtt{d})$. We obtain $(\!|\mathtt{e}|\!) > (\!|\mathtt{e}'|\!)$ by definition of an interpretation and since $(\!|\cdot|\!) > (\!|\cdot|\!)$ is stable by substitution.
- the (t)-rule, then $(\!|\mathtt{t\ e}_1 \ldots \mathtt{e}_i \ldots\ \mathtt{e}_n|\!) > (\!|\mathtt{t\ e}_1 \ldots \mathtt{e}'_i\ \ldots\ \mathtt{e}_n|\!)$ is obtained by definition of $>$ and since $(\!|\mathtt{t}|\!)$ is monotonic, according to Remark 3.
- the $(:)$-rule, then $\mathtt{e} = \mathtt{d} : \mathtt{d}_0$ and $\mathtt{e}' = \mathtt{d}' : \mathtt{d}_0$, for some $\mathtt{d}, \mathtt{d}'$ such that $\mathtt{d} \to \mathtt{d}'$. By induction hypothesis, $(\!|\mathtt{d}|\!) > (\!|\mathtt{d}'|\!)$, so:

$$\begin{aligned} \forall Z \geq 1, (\!|\mathtt{d} : \mathtt{d}_0|\!)(Z) &= 1 + (\!|\mathtt{d}|\!) + (\!|\mathtt{d}_0|\!)(Z-1) \\ &> 1 + (\!|\mathtt{d}'|\!) + (\!|\mathtt{d}_0|\!)(Z-1) = (\!|\mathtt{d}' : \mathtt{d}_0|\!)(Z) \end{aligned}$$

Notice that it also works for the base case $Z = 0$ by definition of $(\!|:|\!)$.

Proposition 2. *Given a closed expression* $\mathtt{e} :: \mathtt{Tau}$ *of a program with an interpretation* $(\!|-|\!)$*, if* $\mathtt{e} \to^n \mathtt{e}'$ *then* $n \leq (\!|\mathtt{e}|\!)$*. In other words, every reduction chain starting from an expression* $\mathtt{e}$ *of a program with interpretation has its length bounded by* $(\!|\mathtt{e}|\!)$.

Corollary 1. *Given a closed expression* $\mathtt{e} :: [\mathtt{Tau}]$ *of a program with an interpretation* $(\!|-|\!)$*, if* $\mathtt{e\ !!\ k} \to^n \mathtt{e}'$ *then* $n \leq (\!|\mathtt{e\ !!\ k}|\!) = (\!|\mathtt{e}|\!)((\!|\mathtt{k}|\!)) = (\!|\mathtt{e}|\!)(k+1)$*, i.e. at most* $(\!|\mathtt{e}|\!)(k+1)$ *reduction steps are needed to compute the* k^{th} *element of a stream* $\mathtt{e}$*.*

Productive streams are defined in the literature [20] as terms weakly normalizing to infinite lists, which is in our case equivalent to:

Definition 12 (Productive stream). A stream `s` is productive if for all value `n :: Nat`, `s !! n` evaluates to a strict value.

Corollary 2. *A closed stream expression admitting an interpretation is productive.*

Proof. This is a direct application of Corollary 1.

Corollary 3. *Given a function symbol* $\mathtt{f :: [Tau]}^k \mathtt{\ ->\ Tau}^l \mathtt{\ ->\ Tau}$ *of a program with a* polynomial *interpretation* $(\!|-|\!)$*, there is a second order polynomial* P *such that if* $\mathtt{f\ e}_1 \dots \mathtt{e}_{k+l} \to_!^n v$ *then* $n \leq P((\!|\mathtt{e}_1|\!), \dots, (\!|\mathtt{e}_{k+l}|\!))$*, for all closed expressions* $\mathtt{e}_1, \dots, \mathtt{e}_{k+l}$*. This polynomial is precisely* $(\!|\mathtt{f}|\!)$*.*

The following lemma shows that in an interpreted program, the number of evaluated stream elements is bounded by the interpretation.

Lemma 5. *Given a function symbol* $\mathtt{f :: [Tau]}^k \mathtt{\ ->\ Tau}^l \mathtt{\ ->\ Tau}$ *of a program with interpretation* $(\!|-|\!)$*, and closed expressions* $\mathtt{e}_1, \dots, \mathtt{e}_l :: \mathtt{Tau}, \mathtt{s}_1, \dots, \mathtt{s}_k :: \mathtt{[Tau]}$*, if* $\mathtt{f\ s}_1 \dots \mathtt{s}_k\ \mathtt{e}_1 \dots \mathtt{e}_l \to_!\ \mathtt{v}$ *and* $\forall \mathtt{n} :: \mathtt{Nat},\ \mathtt{s}_i\ \mathtt{!!\ n} \to_!\ \mathtt{v}_i^{\mathtt{n}}$ *then for all closed expressions* $\mathtt{s}'_1 \dots \mathtt{s}'_k :: [\mathtt{Tau}]$ *satisfying:*

$$\forall \mathtt{n} :: \mathtt{Nat}\ \textit{if}\ (\!|\mathtt{n}|\!) \leq (\!|\mathtt{f\ s}_1 \dots \mathtt{s}_k\ \mathtt{e}_1 \dots \mathtt{e}_l|\!) + \mathtt{1}\ \textit{then}\ \mathtt{s}'_\mathtt{i}\ \mathtt{!!\ n} \to_!\ \mathtt{v}_i^{\mathtt{n}}$$

we have:

$$\mathtt{f\ s}'_1 \dots \mathtt{s}'_k\ \mathtt{e}_1 \dots \mathtt{e}_l \to_!\ \mathtt{v}.$$

Proof. Let $N = (\!|\mathtt{f\ s}_1 \dots \mathtt{s}_k\ \mathtt{e}_1 \dots \mathtt{e}_l|\!)$. Since pattern matching on stream arguments has depth 1, the N^{th} element of a stream cannot be evaluated in less than N steps. $\mathtt{f\ s}_1 \dots \mathtt{s}_k\ \mathtt{e}_1 \dots \mathtt{e}_l$ evaluates in less than N steps, so at most the first N elements of the input stream expressions can be evaluated, so the reduction steps of $\mathtt{f\ s}_1 \dots \mathtt{s}_k\ \mathtt{e}_1 \dots \mathtt{e}_l$ and $\mathtt{f\ s}'_1 \dots \mathtt{s}'_k\ \mathtt{e}_1 \dots \mathtt{e}_l$ are exactly the same.

5. Characterizations of polynomial time

In this section, we provide a characterization of polynomial time UOTM computable functions using interpretations. We also provide a partial characterization of Basic Feasible Functionals using the same methodology.

Definition 13. The expressions in a program represent integers, functions, or type-2 functionals in the following sense:

- an expression `e :: Bin` computes an integer n if `e` evaluates to a strict value representing the binary encoding of n.
- an expression `e :: [Bin]` computes a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ if `e !! n` computes $f(n)$.
- a function symbol `f` computes a function $F : (\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow \mathbb{N}$ if $\mathtt{f}\ \mathtt{s}_1\ \ldots \mathtt{s}_k\ \mathtt{e}_1\ \ldots\ \mathtt{e}_l$ computes $F(g_1, \ldots g_k, x_1, \ldots x_l)$ for all expressions $\mathtt{s}_1, \ldots\ \mathtt{s}_k, \mathtt{e}_1, \ldots\ \mathtt{e}_l$ of respective types [`Bin`] and `Bin` computing some functions $g_1, \ldots g_k$ and integers $x_1, \ldots x_l$.

Theorem 2. *A function $F : (\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow \mathbb{N}$ is computable in polynomial time by a* UOTM *if and only if there exists a program* `f` *computing F, of type* $\mathtt{[Bin]}^k$ `->` $\mathtt{Bin}^l$ `->` `Bin` *which admits a polynomial interpretation.*

To prove this theorem, we will demonstrate in Lemma 6 that second order polynomials can be computed by programs having polynomial interpretations. We will then use this result to get completeness in Lemma 7. Soundness (Lemma 9) consists in computing a bound on the number of inputs to read in order to compute an element of the output stream and then to perform the computation by a classical Turing machine.

5.1. Completeness

Lemma 6. *Every second order polynomial can be computed in unary by a polynomial program.*

Proof. Examples 6 and 7 give polynomial interpretations of unary addition (`plus`) and multiplication (`mult`) on unary integers (`Nat`). Then, we can define a program `f` computing the second order polynomial P by $\mathtt{f}\ \mathtt{y}_1 \ldots \mathtt{y}_k\ \mathtt{x}_1 \ldots \mathtt{x}_l = \mathtt{e}$ where `e` is the strict implementation of P:

- X_i is implemented by the zero order variable $\mathtt{x_i}$.
- $Y_i\langle P_1\rangle$ is computed by $\mathtt{y_i}\ \mathtt{!!}\ \mathtt{e}_1$, if $\mathtt{e}_1$ computes P_1.
- The constant $n \in \mathbb{N}$ is implemented by the corresponding strict value $\mathtt{n} :: \mathtt{Nat}$.
- $P_1 + P_2$ is computed by $\mathtt{plus}\ \mathtt{e}_1\ \mathtt{e}_2$, if $\mathtt{e}_1$ and $\mathtt{e}_2$ compute P_1 and P_2 respectively.
- $P_1 \times P_2$ is computed by $\mathtt{mult}\ \mathtt{e}_1\ \mathtt{e}_2$, if $\mathtt{e}_1$ and $\mathtt{e}_2$ compute P_1 and P_2 respectively.

Since `plus`, `mult` and `!!` have a polynomial interpretation, $(\!|\mathtt{e}|\!)$ is a second order polynomial $P_\mathtt{e}$ in $(\!|\mathtt{y}_1|\!), \ldots (\!|\mathtt{y}_k|\!), (\!|\mathtt{x}_1|\!), \ldots, (\!|\mathtt{x}_l|\!)$ and we just set $(\!|\mathtt{f}|\!) = \mathtt{P_e} + \mathtt{1}$.

Lemma 7 (Completeness). *Every polynomial time* UOTM *computable function can be computed by a polynomial program.*

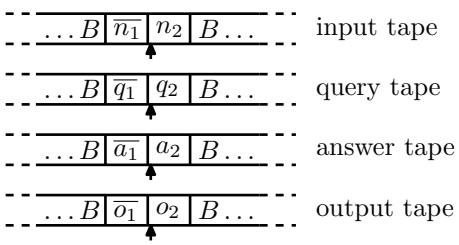

Figure 5: Encoding of the content of the tapes of an OTM (or UOTM). $\overline{w}$ represents the mirror of the word w and the arrows represent the positions of the heads.

PROOF. Let $f : (\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow \mathbb{N}$ be a function computed by a UOTM $\mathcal{M}$ in time P, with P a second order polynomial. Without loss of generality, we will assume that $k = l = 1$. The idea of this proof is to write a program whose function symbol $\mathtt{f_0}$ computes the output of $\mathcal{M}$ after t steps, and to use Lemma 6 to simulate the computation of P.

Let $\mathtt{f_0}$ be the function symbol describing the execution of $\mathcal{M}$:

$$\mathtt{f_0 :: [Bin] -> Nat -> Nat -> Bin^8 -> Bin}$$

The arguments of $\mathtt{f_0}$ represent respectively the input stream, the number of computational steps $\mathtt{t}$, the current state and the 4 tapes (each tape is represented by two binary numbers as illustrated in Figure 5). The output will correspond to the content of the output tape after $\mathtt{t}$ steps.

The function symbol $\mathtt{f_0}$ is defined recursively in its second argument:

- if the timer is 0, then the program returns the content of the output tape (after its head):

$$\mathtt{f_0\ s\ 0\ q\ n_1\ n_2\ q_1\ q_2\ a_1\ a_2\ o_1\ o_2\ =\ o_2}$$

- for each transition of $\mathcal{M}$, we write a definition of this form:

$$\mathtt{f_0\ s\ (t+1)\ q\ n_1\ n_2\ q_1\ q_2\ a_1\ a_2\ o_1\ o_2}$$
$$\mathtt{= f_0\ s\ t\ q'\ n_1'\ n_2'\ q_1'\ q_2'\ a_1'\ a_2'\ o_1'\ o_2'}$$

  where $\mathtt{n_1}$ and $\mathtt{n_2}$ represent the input tape before the transition and $\mathtt{n_1'}$ and $\mathtt{n_2'}$ represent the input tape after the transition, the motion and writing of the head being taken into account, and so on for the other tapes.

Since the transition function is well described by a set of such definitions, the function $\mathtt{f_0}$ produces the content of $\mathtt{o_2}$ (*i.e.* the content of the output tape) after t steps on input $\mathtt{t}$ and configuration $\mathcal{C}$ (*i.e.* the state and the representations of the tapes).

$\mathtt{f}_0$ admits a polynomial interpretation $(\!|\mathtt{f}_0|\!)$. Indeed, in each definition, the state can only increase by a constant, the length of the numbers representing the various tapes cannot increase by more than 1. The answer tape $(\!|\mathtt{a}_2|\!)$ can undergo an important increase: when querying, it can increase by $(\!|\mathtt{s}|\!)((\!|\mathtt{q}_2|\!))$, that is the interpretation of the input stream taken in the interpretation of the query.

Then we can provide a polynomial interpretation to $\mathtt{f}_0$:

$$(\!|\mathtt{f}_0|\!)(Y, T, Q, N_1, N_2, Q_1, Q_2, A_1, A_2, O_1, O_2) = \\ (T+1) \times (Y\langle Q_2\rangle + 1) + Q + N_1 + N_2 + Q_1 + A_1 + A_2 + O_1 + O_2$$

Lemma 6 shows that the polynomial P can be implemented by a program `p` with a polynomial interpretation. Finally, consider the programs `size`, `max`, `maxsize` and $\mathtt{f}_1$ defined below:

```
size :: Bin -> Nat
size Nil = 0
size (0 x) = (size x)+1
size (1 x) = (size x)+1

max :: Nat -> Nat -> Nat
max 0 0 = 0
max 0 (k+1) = k+1
max (n+1) 0 = n+1
max (n+1) (k+1) = (max n k)+1

maxsize :: [Bin] -> Nat -> Nat
maxsize (h:t) 0 = size h
maxsize (h:t) (n+1) = max (maxsize t n) (size h)

f1 :: [Bin] -> Bin -> Bin
f1 s n = f0 s (p (maxsize s) (size n)) q0 Nil n Nil ... Nil
```

where $\mathtt{q}_0$ is the index of the initial state, `size` computes the size of a binary number, and `maxsize` computes the size function of a stream of binary numbers. $\mathtt{f}_1$ computes an upper bound on the number of steps before $\mathcal{M}$ halts on input n with oracle s (*i.e.* $P(|s|, |n|)$), and simulates $\mathtt{f}_0$ within this time bound. The output is then the value computed by $\mathcal{M}$ on these inputs. Define the following polynomial interpretations for `max`, `size` and `maxsize`:

$$(\!|\mathtt{size}|\!)(X) = 2X$$
$$(\!|\mathtt{max}|\!)(X_1, X_2) = X_1 + X_2$$
$$(\!|\mathtt{maxsize}|\!)(Y, X) = 2 \times Y\langle X\rangle$$

Finally $\mathtt{f}_1$ admits a polynomial interpretation since it is defined by composition of programs with polynomial interpretations.

Adapting the proof of Lemma 7 in the case of an UOTM without type-1 input (*i.e.* an ordinary Turing machine) allows us to state a similar result for type-1 functions (which can already be deduced from well known results in the literature).

Corollary 4. *Every polynomial time computable type-*1 *function can be computed by a polynomial stream-free program.*

5.2. Soundness

In order to prove the soundness result, we need to prove that the polynomial interpretation of a program can be computed in polynomial time by a UOTM in this sense:

Lemma 8. *If P is a second-order polynomial, then the function:*

$$F_1, \ldots, F_k, x_1, \ldots x_l \mapsto 2^{P(|F_1|,\ldots,|F_k|,|x_1|,\ldots,|x_l|)} - 1$$

is computable in polynomial time by a UOTM.

PROOF. The addition and multiplication on unary integers, and the function $x \mapsto |x|$ are clearly computable in polynomial time. Polynomial time is also stable under composition, so we only need to prove that the size function $|F|$ is computable in polynomial time by a UOTM. This is the case since it is a max over a polynomial number of elements of polynomial length. Note that this would not be true with the size function $||.||$ as defined in [3] since is not computable in polynomial time.

Lemma 9 (Soundness). *If a function symbol* `f :: [Bin]`^{k} `-> Bin`^{l} `-> Bin` *admits a polynomial interpretation, then it computes a type-*2 *function f of type $(\mathbb{N} \longrightarrow \mathbb{N})^k \longrightarrow \mathbb{N}^l \longrightarrow \mathbb{N}$ which is computable in polynomial time by a* UOTM.

PROOF. From the initial program (which uses streams), we can build a program using finite lists instead of streams as follows.

For each inductive type `Tau`, let us define the inductive type of finite lists over `Tau`:

```
data List(Tau) = Cons Tau List(Tau) | []
```

The type of each function symbol is changed from `[Bin]`^{k} `-> Bin`^{l} `-> Bin` to `List(Bin)`^{k} `-> Bin`^{l} `-> Nat -> Bin` (or from `[Bin]`^{k} `-> Bin`^{l} `-> [Bin]` to `List(Bin)`^{k} `-> Bin`^{l} `-> Nat -> List(Bin)`): streams are replaced by lists, and there is an extra unary argument. We also add an extra constructor `Err` to each type definition.

For each definition in the program, we replace the stream constructor (`:`) with the list constructor (`Cons`). We also add extra definitions matching the cases where some of the list arguments match the empty list (`[]`). In this case, the left part is set to `Err`. Whenever a function is applied to this special value, it returns it. This defines a new program with only inductive types, which behaves similarly to the original one.

The following program transforms a finite list of binary words into a stream by completing it with zeros.

```
app :: List(Bin) -> [Bin]
app (Cons h t) = h : (app t)
app [] = 0 : (app [])
```

It is easy to verify that $(\!|\texttt{app}|\!)(\texttt{X})(\texttt{Z}) = \texttt{X} + \texttt{3Z}$ is a correct interpretation. On strict values, the new program:

$$\texttt{f'}\ \texttt{v}_1\ \ \ldots\ \texttt{v}_k\ \texttt{v}_{k+1}\ \texttt{v}_{k+l}$$

reduces as the original one with lists completed with `app`:

$$\texttt{f}\ (\texttt{app}\ \texttt{v}_1)\ \ \ldots\ (\texttt{app}\ \texttt{v}_k)\ \texttt{v}_{k+1}\ \texttt{v}_{k+l}.$$

The only differences are the additional reductions steps for `app`, but there is at most one additional reduction step for each stream argument each time there is a definition reduction ((d) rule in Figure 4), so the total number of reductions is at most multiplied by a constant. The evaluation may also terminate earlier if the error value appears at some point.

The number of reduction steps is then bounded (up to a multiplicative constant) by:

$$(\!|\texttt{f}|\!)((\!|\texttt{app}\ \texttt{v}_1|\!), \ldots (\!|\texttt{app}\ \texttt{v}_k|\!), (\!|\texttt{v}_{k+1}|\!), \ldots (\!|\texttt{v}_{k+l}|\!))$$

Since $(\!|\texttt{app}\ \texttt{v}_i|\!)$ is a first order polynomial in the interpretation of $\texttt{v}_i$, Lemma 1 proves that the previous expression is a first order polynomial in the interpretation of its inputs. Several results in the literature (for example [21, 22]) on type-1 interpretations allow us to state that this new program computes a polynomial type-1 function (in particular because it is an orthogonal term rewriting system).

Let us now build a UOTM which computes `f`. According to Lemma 8, given some inputs and oracles, we can compute $(\!|\texttt{f}|\!)$ applied to their sizes and get a unary integer N in polynomial time. The UOTM then computes the first N values of each type-1 input to obtain finite lists (of polynomial size) and then compute the corresponding list function on these inputs. According to Lemma 5, since the input lists are long enough, the result computed by the list function and the stream function are the same.

5.3. Basic feasible functionals

In the completeness proof (Lemma 7), the program built from the UOTM deals with streams using only the `!!` function to simulate oracle calls. This translation can be easily adapted to implement OTMs with stream programs. Because of the strict inclusion between polynomial time UOTM computable functions and BFF (*cf.* Lemma 2) and the Lemma 9, it will not always be possible to provide a polynomial interpretation to this translation, but a new proof provides us with a natural class of interpretation functions.

Definition 14 (exp-poly). Let exp-poly be the set of functions generated by the following grammar:

$$EP := P \mid EP + EP \mid EP \times EP \mid Y\langle 2^{EP}\rangle$$

The interpretation of a program is exp-poly if each symbol is interpreted by an exp-poly function.

Example 10. $P(Y, X) = Y\langle 2^{X^2+1}\rangle$ is in exp-poly, whereas $2^X \times Y\langle 2^X\rangle$ is not.

Theorem 3. *Every* BFF *functional is computed by a program which admits an exp-poly interpretation.*

PROOF. Let f be a function computed by an OTM $\mathcal{M}$. We can reuse the construction from the proof of lemma 7 to generate a function symbol $\mathtt{f_0}$ (and its definitions) simulating the machine. Since we consider an OTM and no longer a UOTM, we should note that when querying, the query tape now contains a binary word, which we have to convert into unary before giving it to the `!!` function using an auxiliary function `natofbin`:

```
natofbin :: Bin -> Nat
natofbin Nil = 0
natofbin (0 x) = plus (natofbin x) (natofbin x)
natofbin (1 x) = (plus (natofbin x) (natofbin x)) +1
```

Its interpretation should verify, using the interpretation of `plus` given in Example 6:

$$(\!|\mathtt{natofbin}|\!)(X+1) > 3 \times (\!|\mathtt{natofbin}|\!)(X) + 1$$

This can be fulfilled with $(\!|\mathtt{natofbin}|\!)(X) = 2^{2X}$. Note that this conversion function has no implementation with sub-exponential interpretation since the size of its output is exponential in the size of the input.

Now, in the new program, oracle calls are translated into `s !! (natofbin a2)`, and the increase in $(\!|\mathtt{a2}|\!)$ is now bounded by $(\!|\mathtt{s}|\!)(2^{(\!|2\mathtt{q_2}|\!)})$.

We can define the interpretation of $\mathtt{f_0}$ by:

$$(\!|\mathtt{f_0}|\!)(Y, T, \dots, Q_2, \dots) = (T+1)\times(Y\langle 2^{2Q_2}\rangle+1)+Q+N_1+N_2+Q_1+A_1+A_2+O_1+O_2$$

Exp-poly functions are also computed by exp-poly programs using the same composition of `natofbin` and `!!`. Finally, the adaptation of the initial proof shows that $\mathtt{f_1}$ will also have an exp-poly interpretation.

Remark 4. The soundness proof (Lemma 9) does not adapt to BFF and exp-poly programs, because the required analogue of Lemma 8 (where P is an exp-poly and the function is computable by a polynomial time OTM) is false (in particular, $x, F \mapsto 2^{2^{F(x)}}$ is not basic feasible). Still, we conjecture that the converse of Theorem 3 holds. That is exp-poly programs only compute BFF functionals.

6. Link with polynomial time computable real functions

We show in this section that our complexity results can be adapted to real functions.

Until now, we have considered stream programs as type-2 functionals in their own rights. However, type-2 functionals can be used to represent real functions. Indeed Recursive Analysis models computation on reals as computation on converging sequences of rational numbers [23, 17]. Note that there are numerous other possible applications, for example Kapoulas [24] uses UOTM to study the complexity of p-adic functions and the following results could be adapted, since p-adic numbers can be seen as streams of integers between 1 ands $p-1$.

We will require a given convergence speed to be able to compute effectively. A real x is represented by a sequence $(q_n)_{n\in\mathbb{N}} \in \mathbb{Q}^{\mathbb{N}}$ if:

$$\forall i \in \mathbb{N}, |x - q_i| < 2^{-i}.$$

This will be denoted by $(q_n)_{n\in\mathbb{N}} \rightsquigarrow x$. In other words, after a query of size n, an UOTM obtains an encoding of a rational number q_n approaching its input with precision 2^{-n}.

Definition 15 (Computable real function). A function $f : \mathbb{R} \longrightarrow \mathbb{R}$ will be said to be computed by an UOTM if:

$$(q_n)_{n\in\mathbb{N}} \rightsquigarrow x \Rightarrow (\mathcal{M}(q_n))_{n\in\mathbb{N}} \rightsquigarrow f(x). \tag{1}$$

We will restrict to functions over the real interval $[0, 1]$ (or any compact set).

Hence a computable real function will be computed by programs of type `[Q] -> [Q]` in our stream language, where `Q` is an inductive type describing the set of rationals $\mathbb{Q}$. For example, we can define the data type of rationals with a pair constructor `/`:

```
data Q = Bin / Bin
```

Only programs encoding machines verifying the implication (1) will make sense in this framework. Following [17], we can define polynomial complexity of real functions using polynomial time UOTM computable functions.

The following theorems are applications of Theorem 2 to this framework.

Theorem 4 (Soundness). *If a program* `f :: [Q] -> [Q]` *with a polynomial interpretation computes a real function, then this function is computable in polynomial time.*

PROOF. Let us define `g` from `f :: [Q] -> [Q]`:

```
g :: [Q] -> Nat -> Q
g s n = (f s) !! n
```

If `f` has a polynomial interpretation $(\!|\mathtt{f}|\!)$, then `g` admits the polynomial interpretation $(\!|\mathtt{g}|\!)(\mathtt{Y},\mathtt{N}) = (\!|\mathtt{f}|\!)(\mathtt{Y},\mathtt{N}) + \mathtt{1}$ (using the usual interpretation of `!!` defined in Example 8).

The type of rational numbers can be seen as pairs of binary numbers, so Theorem 2 can be easily adapted to this framework. A machine computes a real function in polynomial time if and only if it outputs the n^{th} element of the result

in polynomial with respect to the size of the input (as defined in Definition 4) and n. In this sense, the machine constructed from `g` using Theorem 2 computes the real function computed by `f`.

Example 11. The square function over the real interval $[0,1]$ can be implemented in our stream language, provided we already have a function symbol `bsqr :: Bin -> Bin` implementing the square function over binary integers:

```
Rsqr :: [Q] -> [Q]
Rsqr (h : t) = Ssqr t

Ssqr :: [Q] -> [Q]
Ssqr ((a / b) : t) = (bsqr a / bsqr b) : (Ssqr t)
```

`Ssqr` squares each element of its input stream. `Rsqr` removes the head of its input before applying `Ssqr` because the square function requires one additional precision unit on its input:

$$\forall n \in \mathbb{N}, |q_n - x| \leq 2^{-n} \Rightarrow \forall n \in \mathbb{N}, |q_{n+1}^2 - x^2| \leq 2^{-n}$$

This allows us to prove that the output stream converges at the right speed and that `Rsqr` indeed computes the real square function on $[0,1]$. Now, set the polynomial interpretations:

$$(\!|\texttt{Ssqr}|\!)(\texttt{Y},\texttt{Z}) = 2 \times \texttt{Z} \times (\!|\texttt{bsqr}|\!)(\texttt{Y}\langle\texttt{Z}\rangle)$$

$$(\!|\texttt{Rsqr}|\!)(\texttt{Y},\texttt{Z}) = 2 \times \texttt{Z} \times (\!|\texttt{bsqr}|\!)(\texttt{Y}\langle\texttt{Z}+1\rangle)$$

They are indeed valid:

$$\begin{aligned}(\!|\texttt{Ssqr ((a / b) : t)}|\!)(\texttt{Z}+1) &= 2(Z+1)(\!|\texttt{bsqr}|\!)(2+\texttt{A}+\texttt{B}+\texttt{T}(\texttt{Z})) \\ &> 2 + (\!|\texttt{bsqr}|\!)(\texttt{A}) + (\!|\texttt{bsqr}|\!)(\texttt{B}) + 2\texttt{Z}(\!|\texttt{bsqr}|\!)(\texttt{T}(\texttt{Z}-1)) \\ &= (\!|\texttt{(bsqr a / bsqr b) : (Ssqr t)}|\!)\end{aligned}$$

and

$$\begin{aligned}(\!|\texttt{Rsqr (h : t)}|\!)(\texttt{Z}) &= 2Z(\!|\texttt{bsqr}|\!)(1+\texttt{H}+\texttt{T}(\texttt{Z}-1+1) \\ &> 2Z(\!|\texttt{bsqr}|\!)(\texttt{T}(\texttt{Z})) = (\!|\texttt{Ssqr t}|\!)\end{aligned}$$

Theorem 5 (Completeness). *Any polynomial-time computable real function can be implemented by a polynomial program.*

Proof. Following [17], we can describe any computable real function f by two functions $f_{\mathbb{Q}} : \mathbb{N} \times \mathbb{Q} \longrightarrow \mathbb{Q}$ and $f_m : \mathbb{N} \longrightarrow \mathbb{N}$ where $f_{\mathbb{Q}}(n,q)$ computes an approximation of $f(q)$ with precision 2^{-n}:

$$\forall q, (f_{\mathbb{Q}}(n,q))_{n\in\mathbb{N}} \rightsquigarrow f(q)$$

and f_m is a modulus of continuity of f defined as follows:

$$\forall n, x, y, |x-y| < 2^{-f_m(n)} \Rightarrow |f(x) - f(y)| < 2^{-n}$$

For a polynomial-time computable real function, those f_m and $f_{\mathbb{Q}}$ are discrete functions computable in polynomial time. Corollary 4 ensures that these functions can be implemented by programs $\mathtt{f_Q}$ and $\mathtt{f_m}$ with polynomial interpretations. Then, we can easily derive a program that computes f by first finding which precision on the input is needed (using f_m) and computing with $f_{\mathbb{Q}}$ an approximation of the image of the input.

```
f_aux :: Nat -> [Q] -> [Q]
f_aux n y = (f_Q n (y !! (f_m n))) : (f_aux (n+1) y)
f :: [Q] -> [Q]
f y = f_aux 0 y
```

with `Q` an inductive type representing rational numbers.

We can easily check that these interpretations work:

$$(\!|\mathtt{f_{aux}}|\!)(\mathtt{Y},\mathtt{N},\mathtt{Z}) = (\mathtt{Z}+\mathtt{1}) \times (\mathtt{1} + (\!|\mathtt{f_Q}|\!)(\mathtt{N}+\mathtt{Z},\mathtt{Y}\langle(\!|\mathtt{f_m}|\!)(\mathtt{N}+\mathtt{Z})\rangle))$$

$$(\!|\mathtt{f}|\!)(\mathtt{Y},\mathtt{Z}) = \mathtt{1} + (\!|\mathtt{f_{aux}}|\!)(\mathtt{Y},\mathtt{1},\mathtt{Z})$$

7. Conclusion

We have provided a characterization of polynomial time stream complexity using basic polynomial interpretations and this the first characterization of this kind. More complex and finer interpretation techniques on first order complexity classes (*e.g.* sup-interpretations or quasi-interpretations) could probably be adapted to stream languages.

This work also provides a partial characterization of BFF and shows that it is not the right feasible complexity class for functions over streams. Our framework also adapts well to applications like computable analysis. We have indeed characterized the class of polynomial time real functions, and this is again the first time that this class is characterized using interpretations.

As a whole, this work is a first step toward higher order complexity. We have used second order interpretations, but higher order interpretations could also be used to characterize higher order complexity classes. The main difficulty is that the state of the art on higher order complexity rarely deal with orders higher than two.

References

[1] R. L. Constable, Type two computational complexity, in: Proc. 5th annual ACM STOC, 108–121, 1973.

[2] K. Mehlhorn, Polynomial and abstract subrecursive classes, in: Proceedings of the sixth annual ACM symposium on Theory of computing, ACM New York, NY, USA, 96–109, 1974.

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

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

[5] 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.

[6] R. Ramyaa, D. Leivant, Ramified Corecurrence and Logspace, Electronic Notes in Theoretical Computer Science 276 (0) (2011) 247 – 261, ISSN 1571-0661.

[7] R. Ramyaa, D. Leivant, Feasible Functions over Co-inductive Data, in: Logic, Language, Information and Computation, vol. 6188 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, ISBN 978-3-642-13823-2, 191–203, 2010.

[8] S. Bellantoni, S. A. Cook, A New Recursion-Theoretic Characterization of the Polytime Functions, Computational Complexity 2 (1992) 97–110.

[9] D. Leivant, J.-Y. Marion, Lambda Calculus Characterizations of Poly-Time, Fundam. Inform. 19 (1/2) (1993) 167–184.

[10] G. Bonfante, J.-Y. Marion, J.-Y. Moyen, Quasi-interpretations a way to control resources, Theoretical Computer Science 412 (25) (2011) 2776 – 2796, ISSN 0304-3975.

[11] P. Baillot, U. D. Lago, Higher-Order Interpretations and Program Complexity, in: P. Cégielski, A. Durand (Eds.), CSL, vol. 16 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, ISBN 978-3-939897-42-2, 62–76, 2012.

[12] Z. Manna, S. Ness, On the termination of Markov algorithms, in: Third Hawaii international conference on system science, 789–792, 1970.

[13] D. Lankford, On proving term rewriting systems are Noetherian, Tech. Rep., Mathematical Department, Louisiana Technical University, Ruston, Louisiana, 1979.

[14] J.-Y. Marion, R. Péchoux, Sup-interpretations, a semantic method for static analysis of program resources, ACM Trans. Comput. Logic 10 (4) (2009) 27:1–27:31, ISSN 1529-3785.

[15] M. Gaboardi, R. Péchoux, Upper Bounds on Stream I/O Using Semantic Interpretations, in: E. Grädel, R. Kahle (Eds.), CSL, vol. 5771 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-04026-9, 271–286, 2009.

[16] R. M. Amadio, Synthesis of max-plus quasi-interpretations, Fundamenta Informaticae 65 (1) (2005) 29–60.

[17] K.-I. Ko, Complexity theory of real functions, Birkhauser Boston Inc. Cambridge, MA, USA, 1991.

[18] H. Férée, E. Hainry, M. Hoyrup, R. Péchoux, Interpretation of Stream Programs: Characterizing Type 2 Polynomial Time Complexity, in: Algorithms and Computation, vol. 6506 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, ISBN 978-3-642-17516-9, 291–303, 2010.

[19] R. Péchoux, Synthesis of sup-interpretations: A survey, Theor. Comput. Sci. 467 (2013) 30–52.

[20] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, J. W. Klop, Productivity of stream definitions, Theor. Comput. Sci. 411 (4-5) (2010) 765–782.

[21] G. Bonfante, A. Cichon, J.-Y. Marion, H. Touzet, Algorithms with polynomial interpretation termination proof, Journal of Functional Programming 11 (01) (2001) 33–53.

[22] U. Lago, S. Martini, Derivational Complexity Is an Invariant Cost Model, in: M. Eekelen, O. Shkaravska (Eds.), Foundational and Practical Aspects of Resource Analysis, vol. 6324 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-642-15330-3, 100–113, 2010.

[23] K. Weihrauch, Computable analysis: an introduction, Springer Verlag, 2000.

[24] G. Kapoulas, Polynomially Time Computable Functions over p-Adic Fields, in: J. Blanck, V. Brattka, P. Hertling (Eds.), Computability and Complexity in Analysis, vol. 2064 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-42197-9, 101–118, 2001.