Interpretation of stream programs: characterizing type 2 polynomial time complexity

Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux

Interpretation of stream programs:
characterizing type 2 polynomial time complexity

Hugo Férée[1], Emmanuel Hainry[2,5], Mathieu Hoyrup[3,5], and Romain Péchoux[4,5]

[1] ENS Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France.
[2] Université Henri Poincaré, Nancy-Université, France.
[3] INRIA Nancy - Grand Est, Villers-lès-Nancy, France.
[4] Université Nancy 2, Nancy-Université, France.
[5] LORIA, BP 239, 54506 Vandœuvre-lès-Nancy cedex, France
hugo.feree@ens-lyon.fr,{hainry,hoyrup,pechoux}@loria.fr

Abstract. We study polynomial time complexity of type 2 functionals. For that purpose, we introduce a first order functional stream language. We give criteria, named well-founded, on such programs relying on second order interpretation that characterize two variants of type 2 polynomial complexity including the Basic Feasible Functions (BFF). These characterizations provide a new insight on the complexity of stream programs. Finally, we adapt these results to functions over the reals, a particular case of type 2 functions, and we provide a characterization of polynomial time complexity in Recursive Analysis.

1 Introduction

Stream languages including lazy functional languages like Haskell allows the programmer to represent functionals, functions over functions. From this perspective, they can be understood as a way to simulate type 2 functions. There are many works in the literature that study computability and (polynomial time) complexity of such functions [5,14]. The implicit computational complexity (ICC) community has proposed characterizations of such complexity classes using function algebra and types [9,16,8] or recently as a logic [15] . These results are reminiscent of former characterizations of type 1 polynomial time functions [4,2,12] that led to other ICC works using polynomial interpretations.

Polynomial interpretations [13,11] are a well-known tool used in the termination analysis of first order functional programs for several decades. Variants, like sup-interpretations and quasi-interpretations [3], that allow the programmer to perform program complexity analysis have emerged in the last ten years. One of their drawbacks is that such tools are restricted to first order programs on inductive data types. The paper [7] was a first attempt to adapt such a tool to co-inductive data types and, more precisely, to stream programs. In this paper, we provide a second order variation of this interpretation methodology that fits to stream computation.

It allows us to characterize exactly the set of functions computable in polynomial time by Unary Oracle Turing Machine (UOTM), that is functions computable by machines including oracles where the oracle has only unary input. It can also be used in order to characterize the set of functions computable in polynomial time by Oracle Turing Machine (OTM), that is shown to be equivalent to the BFF algebra in [9].

The first characterization has two advantages. First, it gives a new and intuitive notion of stream computational complexity in terms of Turing Machine. Second, it shows that this natural class can be easily captured using an adaptation of the interpretation tool. Using this tool 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 Turing Machine). The drawback is that the tool suffers from the same problem as polynomial interpretation: the difficulty to automatically synthesize the interpretation of a given program (see [1]).

The latter characterization gives a natural view of a well-know complexity class BFF, just by changing the interpretation codomain: indeed we use power towers instead of polynomials in the interpretation of a stream argument. It illustrates that the first characterization on UOTM is natural and flexible because it can be easily adapted to other complexity classes. Finally, it can be interpreted as a negative result showing that the BFF class, whose purpose is to study functions from $\mathbb{N} \to \mathbb{N}$, is surprisingly not well-suited to describe stream polynomial complexity (because of the power tower).

We also go one step further showing that these tools can be adapted to characterize the complexity of functions computing over reals defined in Recursive Analysis [10]. This approach is a first attempt to study the complexity of such functions through static analysis methods.

Outline of the paper The paper is organized as follows. In section 2, we briefly recall the notion of (Unary) Oracle Turing Machine and its complexity. In section 3, we introduce a first order stream language. In section 4, we define the interpretation tools and a criterion on stream programs. We show our main characterization relying on the criterion in section 5. In a last section, this characterization is adapted to functions computing over reals.

2 Polynomial time Oracle Turing Machines

In this section, we recall the notion of Oracle Turing Machine, used by Kapron and Cook in their characterization of Basic Poly-time functionals (BFF) [9], and we give a variant, Unary Oracle Turing Machine, more related to stream computations.

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

If $\mathcal{M}$ is used with oracles $F_1, \dots F_k : \mathbb{N} \to \mathbb{N}$, then on the oracle state $i \in \{1, \dots, k\}$, $F_i(x)$ is written on the corresponding answer tape, whenever x is the content of the corresponding query tape.

We introduce now the notion of Unary OTM that are 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 1 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, $F_i(|x|)$ is written on the corresponding answer tape, whenever x is the content of the corresponding query tape.*

Definition 3 (Size of function). *The size $|F| : \mathbb{N} \to \mathbb{N}$ of a function $F : \mathbb{N} \to \mathbb{N}$ is defined by:*

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

where $|F(k)|$ represents the size of the binary representation of $F(k)$.

Definition 4 (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 first order variable, Y a second order one and c a constant in $\mathbb{N}$.

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

Definition 5 (Polynomial running time). *The cost of a transition is:*

- *$|F|(|x|)$, if the machine is in a query state of the oracle F on input query x;*[1]
- *1 otherwise.*

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

A function $G : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ is OTM computable (resp. UOTM computable) in polynomial time if there exists a second order polynomial P such that $G(F_1, \dots, F_k, x_1, \dots x_l)$ is computed by an OTM (resp. UOTM) in time $P(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)$ on inputs $x_1, \dots, x_l$ and oracles $F_1, \dots, F_k$.

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 in [9]):

[1] This definition is equivalent to that of [9] which considers $|F|(x)$ but where $|F|$ is the maximum for $k \leq |n|$. It has the advantage to be uniform for UOTM and OTM.

Example 1. The function $G : (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$ defined by $G(F, x) = F(|x|)$ is UOTM computable in polynomial time (its running time is bounded by $2 \times (|x| + |F|(|x|))$, that is the cost to copy x on the query tape and to query the oracle). However $H(F, x) = F(x)$ is not but is in BFF.

3 First order stream language

Syntax. We define a simple first order functional language with streams. This small language can be seen as a strict subset of a lazy functional language like Haskell. We denote by $\mathcal{F}$ the set of function symbols, $\mathcal{C}$ the set of constructor symbols and $\mathcal{X}$ the set of variable names. Programs in our language are lists of definitions $\mathcal{D}$ given by the following grammar:

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

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 $\mathtt{hd} : \mathtt{tl}$, $\mathtt{hd}$ is called the head and $\mathtt{tl}$ is called the tail (of the stream).

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. It 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). 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.

Type system. Programs contain inductive types that will be denoted by `Tau` throughout the paper. Unary integers are defined by `data Nat = 0 | Nat +1`, given that $\mathtt{0}, \mathtt{+1} \in \mathcal{C}$. 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` $\to$ `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).

Each function symbol `f` comes with a typed signature that we restrict to be either $\mathtt{f} :: [\mathtt{Tau}]^k \to \mathtt{Tau}^l \to \mathtt{Tau}$ or $\mathtt{f} :: [\mathtt{Tau}]^k \to \mathtt{Tau}^l \to [\mathtt{Tau}]$, 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 following simple rules with $\mathtt{A}, \mathtt{A}_i \in \{\mathtt{Tau}, \mathtt{[Tau]}\}$:

$$\frac{}{\mathtt{x} :: \mathtt{A}}\ \mathtt{x} \in \mathcal{X} \qquad \frac{\mathtt{t} :: \mathtt{A}_1 \to \ldots \to \mathtt{A}_n \to \mathtt{A} \qquad \forall i \in \{1,n\},\ \mathtt{e}_i :: \mathtt{A}_i}{\mathtt{t}\ \mathtt{e}_1 \ldots \mathtt{e}_n :: \mathtt{A}}\ \mathtt{t} \in \mathcal{C} \cup \mathcal{F}$$

Semantics. Let lazy values and strict values be defined by:

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

where $\mathtt{e}_1, \mathtt{e}_2$ are closed expressions and $\mathtt{c}$ belongs to $\mathcal{C} \setminus \{:\}$. Lazy values are expressions with the constructor symbol : at the top level whereas strict values are expressions 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 derivation rules are defined by:

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

$$\frac{\mathtt{e}_i \to \mathtt{e}_i' \qquad \mathtt{t} \in \mathcal{F} \cup \mathcal{C} \setminus \{:\}}{\mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i\ \ldots\ \mathtt{e}_n \to \mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i'\ \ldots\ \mathtt{e}_n}\ (t) \qquad \frac{\mathtt{e} \to \mathtt{e}'}{\mathtt{e} : \mathtt{e}_0 \to \mathtt{e}' : \mathtt{e}_0}\ (:)$$

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 derivation 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 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 we do not allow stream patterns of depth greater than 1 in a definition).

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

4 Second order polynomial interpretations

In the following, we call a positive functional any function in $((\mathbb{N} \to \mathbb{N})^k \times \mathbb{N}^l) \to T$ with $k, l \in \mathbb{N}$ and $T \in \{\mathbb{N}, \mathbb{N} \to \mathbb{N}\}$. Given a positive functional $F : ((\mathbb{N} \to \mathbb{N})^k \times \mathbb{N}^l) \to T$, the arity of F is $k + l$.

Let $>$ denote the usual ordering on $\mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$, *i.e.* given $F, G : \mathbb{N} \to \mathbb{N}$, $F > G$ if $\forall n \in \mathbb{N} \setminus \{0\}, F(n) > G(n)$. We extend this ordering to positive functionals of arity l by: $F > G$ if $\forall x_1 \dots x_l \in \{\mathbb{N} \setminus \{0\}, \mathbb{N} \to^{\uparrow} \mathbb{N}\}$, $F(x_1, \dots, x_l) > G(x_1, \dots, x_l)$, where $\mathbb{N} \to^{\uparrow} \mathbb{N}$ is the set of increasing functions on positive integers.

Definition 6 (Monotonic positive functionals). *A positive functional F of arity n is monotonic if $\forall i \in \{1, n\}, \forall x_i > x'_i,\ F(\dots, x_i, \dots) > F(\dots, x'_i, \dots)$, where $x_i, x'_i \in \{\mathbb{N} \setminus \{0\}, \mathbb{N} \to^{\uparrow} \mathbb{N}\}$.*

Definition 7. *The interpretation of a program is a total mapping of the function and constructor symbols to monotonic positive functionals. The type of the interpretation is inductively defined by the type of the corresponding symbol:*

- *a symbol* `t` *of type* `Tau` *has interpretation* $(\!|\mathtt{t}|\!)$ *in* $\mathbb{N}$
- *a symbol* `t` *of type* `[Tau]` *has interpretation* $(\!|\mathtt{t}|\!)$ *in* $\mathbb{N} \to \mathbb{N}$
- *a symbol* `t` *of type* `A -> B` *has interpretation* $(\!|\mathtt{t}|\!)$ *in* $T_A \to T_B$*, where* T_A *and* T_B *are the types of the interpretations of the symbols of type* `A` *and, respectively, type* `B`*.*

We fix the interpretation of each constructor symbol by:

- $(\!|\mathtt{c}|\!)(X_1, \dots, X_n) = X_1 + \dots X_n + 1$ *if* $\mathtt{c} \in \mathcal{C} \setminus \{:\}$ *is of arity* n
- $(\!|:|\!)(X, Y)(Z + 1) = 1 + X + Y\langle Z \rangle$[2]
- $(\!|:|\!)(X, Y)(0) = X$

Once the interpretation of each function and constructor symbol is fixed, we can define the interpretation of any expression by structural induction (notice that we preserve the previous correspondence between the type of the expression and the type of its interpretation):

- $(\!|\mathtt{x}|\!) = X$ *if* `x` *is a variable of type* `Tau`*, i.e. we associate a unique first 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 second order variable* $Y : \mathbb{N} \to \mathbb{N}$ *to each* $\mathtt{y} \in \mathcal{X}$ *of type* `[Tau]`*.*
- $(\!|\mathtt{t\ e_1\ \dots\ e_n}|\!) = (\!|\mathtt{t}|\!)((\!|\mathtt{e_1}|\!), \dots, (\!|\mathtt{e_n}|\!))$ *if* $\mathtt{t} \in \mathcal{C} \cup \mathcal{F}$

Consequently, the interpretation $(\!|\ |\!)$ maps any expression to a functional (of the interpretation of its free variables).

The interpretation of a program is polynomial if each function symbol is interpreted by a second order polynomial.

Example 2. The stream constructor : has type `Tau -> [Tau] -> [Tau]`. Consequently, its interpretation $(\!|:|\!)$ has type $(\mathbb{N} \times (\mathbb{N} \to \mathbb{N})) \to (\mathbb{N} \to \mathbb{N})$[3]. Considering the expression $\mathtt{p} : (\mathtt{q} : \mathtt{r})$, with $\mathtt{p}, \mathtt{q}, \mathtt{r} \in \mathcal{X}$, we obtain that:

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

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

[2] By abuse of notation, we will consider that the interpretation of : is also a positive functional.

[3] We will use the cartesian product instead of the arrow for the argument types of a function symbol in the following.

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

Lemma 1. *The interpretation of an expression* `e` *defines a positive functional in the interpretations of its free variables.*

Definition 8 (Well-founded polynomial interpretation). *The interpretation of a program is well-founded if for each definition* $\texttt{f p}_1 \dots \texttt{p}_n = \texttt{e} \in \mathcal{D}$,

$$(\!|\texttt{f p}_1 \dots \texttt{p}_n|\!) > (\!|\texttt{e}|\!)$$

By extension, a program is well-founded (polynomial) if it admits a well-founded (polynomial) interpretation.

The following programs are examples of well-founded polynomial programs:

Example 3. The sum and product over unary integers:

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

They admit the following well-founded interpretation[4] $(\!|\texttt{plus}|\!)(X_1,X_2) = 2 \times X_1 + X_2$, $(\!|\texttt{mult}|\!)(X_1,X_2) = 3 \times X_1 \times X_2$. Indeed, we check that the following inequalities are satisfied:

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

`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
```

and admits a well-founded interpretation $(\!|\texttt{!!}|\!)$ in $((\mathbb{N} \to \mathbb{N}) \times \mathbb{N}) \to \mathbb{N}$ defined by $(\!|\texttt{!!}|\!)(Y,N) = Y\langle N \rangle$. Indeed, we check that:

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

In the same way, we let the reader check that `tln`, which drops the first $n+1$ elements of a stream, admits the well-founded interpretation $(\!|\texttt{tln}|\!)$ of type $((\mathbb{N} \to \mathbb{N}) \times \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ defined by $(\!|\texttt{tln}|\!)(Y,N)(Z) = Y\langle N + Z + 1 \rangle$.

[4] On programs without streams, well-founded polynomial interpretations correspond exactly to polynomial interpretations.

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

Indeed, for the first rule, we just check that $(\!|\texttt{tln (h:t) (n+1)}|\!) > (\!|\texttt{tln t n}|\!)$, that is $\forall Z \in \mathbb{N} \setminus \{0\}$, $(\!|\texttt{tln (h:t) (n+1)}|\!)(Z) > (\!|\texttt{tln t n}|\!)(Z)$.

Lemma 2. *If* $\texttt{e}$ *is an expression of a program with a well-founded interpretation* $(\!|\ |\!)$ *and* $\texttt{e} \to \texttt{e}'$*, then* $(\!|\texttt{e}|\!) > (\!|\texttt{e}'|\!)$.

Corollary 1. *Given a closed expression* $\texttt{e} :: \texttt{Tau}$ *of a program having a well-founded interpretation* $(\!|\ |\!)$*, if* $\texttt{e} \to^n \texttt{e}'$ *then* $n \leq (\!|\texttt{e}|\!)$*, i.e. every reduction chain starting from an expression* $\texttt{e}$ *of a well-founded program has its length bounded by* $(\!|\texttt{e}|\!)$.

Corollary 2. *Given a closed expression* $\texttt{e} :: [\texttt{Tau}]$ *of a program having a well-founded interpretation* $(\!|\ |\!)$*, if* $\texttt{e !! k} \to^n \texttt{e}'$ *then* $n \leq (\!|\texttt{e !! k}|\!) = {}^{5}(\!|\texttt{e}|\!)((\!|\texttt{k}|\!)) = (\!|\texttt{e}|\!)(k+1)$*, i.e. at most* $(\!|\texttt{e}|\!)(k+1)$ *reduction steps are needed to compute the* k^{th} *element of a stream* $\texttt{e}$*.*

Productive streams are defined in the literature [6] as terms weakly normalizing to infinite lists, which is in our case equivalent to: a stream `s` is productive if for all `n :: Nat`, `s !! n` evaluates to a strict value.

Corollary 3. *Each stream expression of a program with a well-founded interpretation is productive.*

Corollary 4. *Given a function symbol* $\texttt{f} :: [\texttt{Tau}]^{\texttt{k}} \to \texttt{Tau}^{\texttt{l}} \to \texttt{Tau}$ *of a program with a well-founded* polynomial *interpretation* $(\!|\ |\!)$*, there is a second order polynomial* P *such that if* $\texttt{f}\ \texttt{e}_1 \dots \texttt{e}_{k+1} \to_!^n \texttt{v}$ *then* $n \leq P((\!|\texttt{e}_1|\!), \dots, (\!|\texttt{e}_{k+1}|\!))$*, for all closed expressions* $\texttt{e}_1, \dots, \texttt{e}_{k+1}$*.*

The following lemma shows that in a well-founded program, the number of evaluated stream elements is bounded.

Lemma 3. *Given a function symbol* $\texttt{f} :: [\texttt{Tau}]^{\texttt{k}} \to \texttt{Tau}^{\texttt{l}} \to \texttt{Tau}$ *of a program having a well-founded interpretation* $(\!|\ |\!)$*, and closed expressions* $\texttt{e}_1, \dots, \texttt{e}_\texttt{l} :: \texttt{Tau}, \texttt{d}_1, \dots, \texttt{d}_\texttt{k} :: [\texttt{Tau}]$*, if* $\texttt{f}\ \texttt{d}_1 \dots \texttt{d}_\texttt{k}\ \texttt{e}_1 \dots \texttt{e}_\texttt{l} \to_!\ \texttt{v}$ *and* $\forall \texttt{n} :: \texttt{Nat}$*,* $\texttt{d}_\texttt{i}\ \texttt{!!}\ \texttt{n} \to_!\ \texttt{v}_\texttt{i}^\texttt{n}$ *then for all closed expressions* $\texttt{d}'_1 \dots \texttt{d}'_\texttt{k} :: [\texttt{Tau}]$ *satisfying* $\texttt{d}'_\texttt{i}\ \texttt{!!}\ \texttt{n} \to_!\ \texttt{v}_\texttt{i}^\texttt{n}$*,* $\forall n \leq (\!|\texttt{f}\ \texttt{d}_1 \dots \texttt{d}_\texttt{k}\ \texttt{e}_1 \dots \texttt{e}_\texttt{l}|\!)$*, we have* $\texttt{f}\ \texttt{d}'_1 \dots \texttt{d}'_\texttt{k}\ \texttt{e}_1 \dots \texttt{e}_\texttt{l} \to_!\ \texttt{v}$*.*

5 Characterizations of polynomial time

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

Note that for the sake of simplicity, we implicitly consider that the inductive type `Tau` has an encoding in $\mathbb{N}$.

[5] Using the well-founded interpretation of example 3

Theorem 1. *A function* $F : ((\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l) \to \mathbb{N}$ *which is computable in polynomial time by a UOTM if and only if there exists a program* `f` *computing F, of type* $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ *admits a polynomial well-founded interpretation.*

To prove this theorem, we will show in lemma 4 that second order polynomials can be computed by programs having well-founded polynomial interpretations. We will then use this result to get soundness in lemma 5. Completeness (lemma 7) consists in computing a bound on the number of entries to read to compute an element of the output stream and then to the computation by a classical Turing machine.

Lemma 4. *Every second order polynomial can be computed by a well-founded polynomial program.*

Lemma 5 (Soundness). *Every polynomial time UOTM computable function can be computed by a well-founded polynomial program.*

Proof

Let $f : ((\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l) \to \mathbb{N}$ 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 $\texttt{f}_0$ giving the output of $\mathcal{M}$ after t steps, and to use lemma 4 to simulate the computation of P.

Let $\texttt{f}_0$ be the function symbol describing the execution of $\mathcal{M}$:

$$\texttt{f}_0 \texttt{ :: [Bin] -> Nat -> Nat -> Bin}^8 \texttt{ -> Bin}$$

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

The arguments of $\texttt{f}_0$ represent respectively the input stream, the number of steps `t` the machine is allowed to compute, the state and the 4 tapes (each tape is represented by 2 binary numbers as illustrated in figure 1). The output will correspond to the content of the output tape after `t` steps.

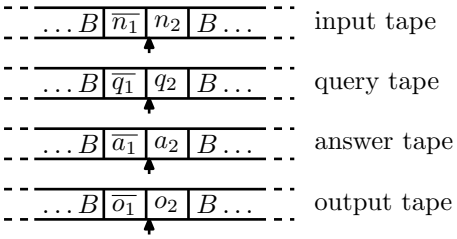

Fig. 1. Encoding of the content of the tapes of an OTM (or UOTM). $\overline{x}$ represents the mirror of the word x and the symbol $\uparrow$ represents the positions of the heads.

The function symbol $\texttt{f}_0$ is defined recursively in its second argument:

– if the timer is 0, then we output the content of the output tape (after its head):
$\texttt{f}_0\ \texttt{s}\ \texttt{0}\ \texttt{q}\ \texttt{n}_1\ \texttt{n}_2\ \texttt{q}_1\ \texttt{q}_2\ \texttt{a}_1\ \texttt{a}_2\ \texttt{o}_1\ \texttt{o}_2\ =\ \texttt{o}_2$

– for each transition of $\mathcal{M}$, we write a definition:
`f0 s (t+1) q n1 n2 q1 q2 a1 a2 o1 o2 = f0 s t q' n1' n2' q1' q2' a1' a2' o1' o2'`
where `n1` and `n2` represent the input tape before the transition and `n1'` and `n2'` 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 `f0` produces the content of `o2` (*i.e.* the content of the output tape) after t steps on entry t and configuration $\mathcal{C}$ (*i.e.* the state and the representations of the tapes).

`f0` admits a well-founded polynomial interpretation $(\!|\texttt{f0}|\!)$. 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 $(\!|\texttt{a2}|\!)$ can undergo an important increase: when querying, it can increase by $(\!|\texttt{s}|\!)((\!|\texttt{q2}|\!))$, that is the interpretation of the input stream taken in the interpretation of the query.

Then, $(\!|\texttt{f0}|\!)(Y, T, Q, N_1, N_2, Q_1, Q_2, A_1, A_2, O_1, O_2)$ can be defined by $(T+1) \times (Y\langle Q_2\rangle + 1) + Q + N_1 + N_2 + Q_1 + A_1 + A_2 + O_1 + O_2$, which provides a well-founded polynomial interpretation. Lemma 4 shows how we can implement the polynomial P by a program `p`, and give it a polynomial well-founded interpretation. Finally, consider the programs `size`, `max`, `maxsize` and `f1` defined below:

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

max :: Nat -> Nat -> Nat
max 0 n = n
max n 0 = n
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 Nil Nil Nil Nil
```

where `q0` 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. `f1` computes an upper bound on the number of steps before $\mathcal{M}$ halts on entry n with oracle s (*i.e.* $P(|s|, |n|)$), and computes `f0` with this time bound. The output is then the value computed by M on these entries. Define the following well-founded polynomial interpretations for `max`, `size` and `maxsize`:

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

Finally `f1` admits a well-founded polynomial interpretation since it is defined by composition of programs with well-founded polynomial interpretations. □

The previous lemma also gives a hint for the completeness proof:

Corollary 5. *Any function $f : \mathbb{N} \to \mathbb{N}$ computable in polynomial time by a Turing Machine can be implemented by a stream program having a well-founded polynomial interpretation.*

Lemma 6. *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.

Lemma 7 (Completeness). *If a program* `f` *of type* $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ *admits a well-founded polynomial interpretation, then it computes a function* $f : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ *which is computable in polynomial time by a UOTM.*

Proof

Lemma 6 shows that given some inputs and oracles, a UOTM can compute $(\!|\texttt{f}|\!)$ applied on their sizes and get a unary integer N in polynomial time. According to lemma 3, the Haskell-like program needs at most the first N values of each oracle. Then, we can build a UOTM which queries all these values (in time $\sum_{i \leq N} |f|(N)$, which is polynomial in the size of the inputs and the size of the oracles) and computes f on these finite inputs: we can convert the program `f` into a program working on finite lists (which will also have polynomial time complexity), and according to corollary 5, this program can be computed in polynomial time by a (classical) Turing Machine. □

Similarly, we can obtain a characterization of BFF, that is functions computable in polynomial time by OTM. Instead of using second order polynomials, we will use a larger set of second order functions named exp-poly.

Definition 9 (exp-poly). *We call exp-poly the set of functions generated by the following grammar:*

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

where P denotes a first order polynomial and Y a second order variable.

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

Theorem 2. *BFF is exactly the set of functions that can be computed by programs that admit a well-founded exp-poly interpretation.*

6 Link with polynomial time computable real functions

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

Up to 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 [17,10].

We will require a given convergence speed to be able to compute effectively. A real x is represented by a sequence $(q_n) \in \mathbb{Q}^{\mathbb{N}}$ if $\forall i \in \mathbb{N}, \|x - q_i\| < 2^{-i}$. This will be denoted by $(q_n) \rightsquigarrow x$. A function $f : \mathbb{R} \to \mathbb{R}$ will be said to be computed by a machine $\mathcal{M}$ if

$$(q_n) \rightsquigarrow x \Rightarrow (\mathcal{M}(q_n)) \rightsquigarrow f(x). \quad (1)$$

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 $\mathbb{Q}$. Only programs encoding machines verifying the implication (1) will make sense in this framework. Following [10], we can define polynomial complexity of real functions using polynomial time UOTM computable functions.

Proposition 1. *If a program* `[Q] -> [Q]` *with a well-founded polynomial interpretation computes a real function on compact* $\mathbb{K}$*, then this function is computable in polynomial time.*

Proposition 2. *Any polynomial-time computable real function (defined over* $\mathbb{K}$*) can be implemented by a well-founded polynomial program.*

References

1. Amadio, R.M.: Synthesis of max-plus quasi-interpretations. Fundamenta Informaticae **65**(1) (2005) 29–60
2. Bellantoni, S., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Computational complexity **2**(2) (1992) 97–110
3. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations. Theor. Comput. Sci.. to appear
4. Cobham, A.: The Intrinsic Computational Difficulty of Functions. In: Logic, methodology and philosophy of science III, North-Holland Pub. Co. (1965) 24
5. Constable, R.L.: Type two computational complexity. In: Proc. 5th annual ACM STOC. (1973) 108–121
6. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.W.: Productivity of stream definitions. Theor. Comput. Sci. **411**(4-5) (2010) 765–782
7. Gaboardi, M., Péchoux, R.: Upper Bounds on Stream I/O Using Semantic Interpretations. In: Computer Science Logic, Springer (2009) 271–286
8. Irwin, R.J., Royer, J.S., Kapron, B.M.: On characterizations of the basic feasible functionals (Part I). J. Funct. Program. **11**(1) (2001) 117–153
9. Kapron, B.M., Cook, S.A.: A new characterization of type-2 feasibility. SIAM Journal on Computing **25**(1) (1996) 117–132
10. Ko, K.I.: Complexity theory of real functions. Birkhauser Boston Inc. Cambridge, MA, USA (1991)
11. Lankford, D.: On proving term rewriting systems are noetherien. tech. rep. (1979)
12. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Typed Lambda Calculi and Applications (1993) 274–288
13. Manna, Z., Ness, S.: On the termination of Markov algorithms. In: Third Hawaii international conference on system science. (1970) 789–792
14. Mehlhorn, K.: Polynomial and abstract subrecursive classes. In: Proceedings of the sixth annual ACM symposium on Theory of computing, ACM New York, NY, USA (1974) 96–109
15. Ramyaa, R., Leivant, D.: Feasible functions over co-inductive data. In: WoLLIC. (2010) 191–203
16. Seth, A.: Turing machine characterizations of feasible functionals of all finite types. Feasible Mathematics II (1995) 407–428
17. Weihrauch, K.: Computable analysis: an introduction. Springer Verlag (2000)