Polynomial time computable real functions

Hugo Férée
Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux

Loria, Nancy

Abstract. In this paper, we study computability and complexity of real functions. We extend these notions, already defined for functions over closed intervals or over the real line to functions over particular real open sets and give some results and characterizations, especially for polynomial time computable functions. Our representation of real numbers as sequences of rational numbers allows us to implement real functions in a stream language. We give a notion of second order polynomial interpretation for this language to guarantee polynomial time complexity.

1 Introduction

Different models have been proposed to define computable real numbers and computable real functions and their complexity. Most of them are based on variations of the Turing machine model (see for example Ko [12] and Weihrauch [17] for recursive analysis and Blum et al. [3,4] for the BSS model), but quite different ones also exist, such as the GPAC model [16,6]. Using Turing machine based models allows us to use comparisons with computable type-1 functions (*i.e.* of type $\mathbb{N} \to \mathbb{N}$) to justify the corresponding definitions of computability and complexity in the domain of real functions and provide similar results. For example, many machine independent characterizations of polynomial time computable type-1 functions already exist, using various tools like function algebra [2] (*i.e.* a class of functions containing some basic functions and closed under some operators, like composition and bounded recursion), typed lambda-calculus [13], linear logic [10] or quasi-interpretation [5] or sup-interpretation [14]. In the same way, the GPAC model [16,6] or the function algebra of Bournez and Hainry [7] are examples of characterizations of some classes of computable real functions. It has also been shown [12,17] that (polynomial time) computable real functions can be described using type-1 functions.

All these results consider functions defined over compact intervals and have been extended to functions defined over the whole real line. Some work [18,12] has been done to study computability of functions over some real open sets (mostly bounded open sets), but studying the complexity of such functions leads to challenging issues, including: What kind of real domains make sense to define computable real functions, and polynomial time computable real functions ? Once they are defined, how can we relate these functions to total real functions or to type-1 functions ? We propose some definitions for functions defined over quite general open domains and give answers to these questions.

Real functions can also be seen as particular cases of type-2 functionals (*i.e.* functions with arguments of type-2: $\mathbb{N} \to \mathbb{N}$). Kapron and Cook have found a function algebra (BFF [11]) corresponding to Basic poly-time functionals [8]. But this definition of polynomial time complexity is not really adapted to describe the complexity of real functions. We propose a variation of the Oracle Turing machine model used for Basic poly-time functionals to define another class of type-2 functionals such that its restriction to real functions gives exactly polynomial time computable (total) real functions.

Our representation of real numbers is closed to the notion of streams (*i.e.* infinite lists) defined in a functional language like Haskell. In the same way that one can guarantee that a program computes a polynomial time computable type-1 function by interpreting its functions by polynomials and by checking that this interpretation (extended to any expression) verifies some properties (*e.g.* it decreases when we apply a rule defining a function), we provide a way to ensure that a program with streams computes a polynomial time computable type-2 function by interpreting symbol function symbols as type-2 polynomials (*i.e.* polynomials with type-2 variables). These well interpreted programs compute exactly the polynomial time computable type-2 functions, and it also happens that a small shift in the definition of type-2 polynomials provides an equivalence with the functions of the BFF algebra.

This paper is organized as follows: We first define the notions of computability and complexity of real functions in section 2 as it has been done in [12] and provide a way to describe real functions with type-1 functions. We extend these notions and results to more general open domains in section 3 and

give a characterization of such functions as functions defined over $\mathbb{R}^2$. Then, in section 4, we describe our oracle Turing machine model and define polynomial time computable type-2 functions. We study a simple Haskell-like language which allows us to compute such functions, seeing type-2 arguments as streams. We provide a notion of polynomial interpretation of such programs and prove that well interpreted programs correspond exactly to polynomial-time type-2 functions. Finally, we shows how our Haskell-like language and its well interpreted programs are related to polynomial time computable real functions.

2 Functions over $\mathbb{R}$

2.1 Computability

We will restrict our study to functions over $\mathbb{R}$, but it can be easily extended to $\mathbb{R}^n$. We denote by $\|x\|$ the absolute value of the real number x (which could, for example, be replaced with the euclidean norm in $\mathbb{R}^n$). We have chosen to represent real numbers using sequences of rational numbers. This choice is justified in remark 1.

Definition 1 (Cauchy sequence representation) *$(q_n)_{n\in\mathbb{N}}$ represents $x \in \mathbb{R}$ if $\forall n \in \mathbb{N}, \|x - q_n\| \leq 2^{-n}$.*

We will denote this by $q_n \rightsquigarrow x$ and say that $(q_n)_{n\in\mathbb{N}}$ is a valid input sequence.

This means that the sequence converges to the real number, and that this convergence is effective.

The definitions of computability and complexity over real numbers come naturally:

Definition 2 *A real number x is computable (resp. computable in polynomial time) if it is represented by a computable sequence $(q_n)_{n\in\mathbb{N}}$ (resp. computable in polynomial time with respect to n).*

Since real numbers have an infinite representation, we need a computational model reading and writing infinite inputs:

Definition 3 (Infinite I/O Turing machine (ITM)) *Let Σ be a finite alphabet and $B \notin \Sigma$ be the blank character. We say that a function $F : ((\Sigma^*)^{\mathbb{N}})^k \to (\Sigma^*)^l \to (\Sigma^*)^{\mathbb{N}}$ is computable by an ITM $\mathcal{M}$ if for all $(y^1_n)_{n\in\mathbb{N}}, \dots (y^k_n)_{n\in\mathbb{N}}$ and $x_1, \dots x_l$, $\mathcal{M}$ writes the sequence $F(y^1, \dots, y^k, x_1, \dots, x_l)$ on its (one-way moving) output tape if $y^1, \dots, y^k, x_1, \dots, x_l$ are written on its input tapes as described in figure 1.*

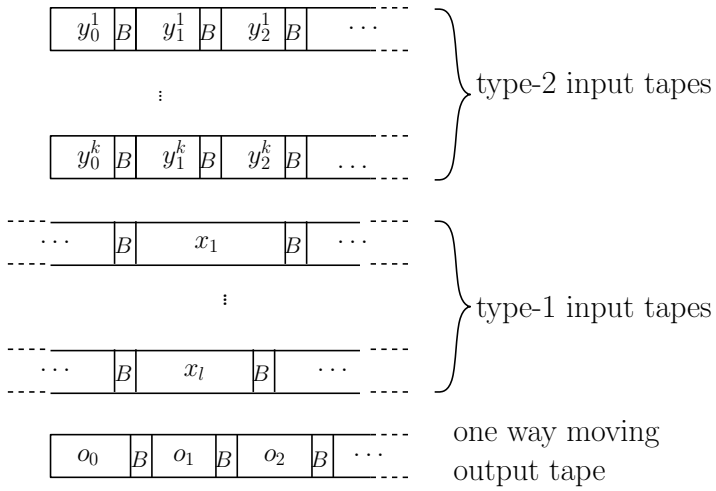

Fig. 1: Infinite I/O Turing machine model

In the following, we will consider functions on rational numbers and sequences of rational numbers. Indeed, we can represent elements of $\mathbb{Q}$ with words on the binary alphabet alphabet $\Sigma = \{0, 1\}$.

Definition 4 *A sequence function $\tilde{f} : \mathbb{Q}^{\mathbb{N}} \to \mathbb{Q}^{\mathbb{N}}$ represents a real function $f : \mathbb{R} \to \mathbb{R}$ if*

$$\forall x \in \mathbb{R}, \forall (q_n)_{n\in\mathbb{N}} \in \mathbb{Q}^{\mathbb{N}}, (q_n)_{n\in\mathbb{N}} \rightsquigarrow x \Rightarrow \tilde{f}((q_n)_{n\in\mathbb{N}}) \rightsquigarrow f(x)$$

Definition 5 *By extension, a real function is computable if it is represented by a computable sequence function.*

A major restriction is that only continuous functions can be computed.

Property 1 *Every computable function is continuous.*

Proof
Proved, for example in [17]. □

Remark 1. One might be surprised that real numbers are not represented using their binary representation. The reason is that many simple functions, like $x \mapsto 3x$ are not computable using this representation. Indeed, if a machine computed this function, then it would decide whether a real number is greater than $1/6$ or not (or would decide the equality on real numbers, which is impossible according to property 1), since the first bit of $3x$ is 1 if and only if $x \geq 1/6$.

This is why we need a redundant representation, *e.g.* using rational numbers represented with two or three binary integers, or a binary representation with signed bits (*i.e.* dyadic numbers) as it is done in [12]. Also see Weihrauch [17] for details about acceptable representations.

In the following, we define K_n as the interval $[-2^n, 2^n]$. We will characterize the computability and complexity of a function using its modulus of continuity on each of these intervals.

Definition 6 *A function $f : \mathbb{R} \to \mathbb{R}$ is described by $f_{\mathbb{Q}} : \mathbb{N} \to \mathbb{Q} \to \mathbb{Q}$ and $f_m : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ if:*

$$P_1 : \forall a, p \in \mathbb{N}, q \in \mathbb{R}, x \in K_a, \|x - q\| \leq 2^{-f_m(a,p)} \Rightarrow \|f(x) - f(q)\| \leq 2^{-(p+1)}$$

$$P_2 : \forall p \in \mathbb{N}, q \in \mathbb{Q}, \|f_{\mathbb{Q}}(p, q) - f(q)\| \leq 2^{-(p+1)}$$

The first property asserts that for all a, $f_m(a, .)$ is the modulus of continuity of f on K_a. The second one stands that on any rational input q, $f(q)$ can be approximated at will by a rational of the form $f_{\mathbb{Q}}(p, q)$.

We claim that (and it is easy to check that) there exists a polynomial time computable function $\gamma : \mathbb{Q} \to \mathbb{N}$ such that: $\forall q \in \mathbb{Q}, q \leq 2^{\gamma(q)-1}$ (which implies that if $(q_n)_{n\in\mathbb{N}} \rightsquigarrow x$, then $x \in K_{\gamma(q_0)}$).

The following functional uses two such functions and γ to compute a real function. $@(f_{\mathbb{Q}}, f_m) : \mathbb{Q}^{\mathbb{N}} \to \mathbb{Q}^{\mathbb{N}}$ is defined by:

$$(q_n)_{n\in\mathbb{N}} \mapsto (f_{\mathbb{Q}}(n, q_{f_m(\gamma(q_0),n)}))_{n\in\mathbb{N}}$$

Property 2 *If f is described by $f_{\mathbb{Q}}$ and f_m, then $@(f_{\mathbb{Q}}, f_m)$ represents f.*

Proof
Let $x \in \mathbb{R}$, $(q_n) \in \mathbb{Q}^{\mathbb{N}}$ such that $(q_n) \rightsquigarrow x$ and $n \in \mathbb{N}$.

$$\|f(x) - (@(f_{\mathbb{Q}}, f_m)(x))_n\| = \|f(x) - f_{\mathbb{Q}}(n, q_{f_m(\gamma(q_0),n)})\|$$

$$\leq \|f(x) - f(q_{f_m(\gamma(q_0),n)})\| + \|f(q_{f_m(\gamma(q_0),n)}) - f_{\mathbb{Q}}(n, q_{f_m(\gamma(q_0),n)})\|$$

by triangular inequality.

By property P_1, since $x \in K_{\gamma(q_0)}$ and $\|x - q_{f_m(\gamma(q_0),n)}\| \leq 2^{-f_m(\gamma(q_0),n)}$ we have $\|f(x) - f(q_{f_m(\gamma(q_0),n)})\| \leq 2^{-(n+1)}$

By property P_2 we get $\|f_{\mathbb{Q}}(q_{f_m(\gamma(q_0),n)}) - f_{\mathbb{Q}}(n, q_{f_m(\gamma(q_0),n)})\| \leq 2^{-(n+1)}$. Summing these two bounds allows us to conclude. □

Corollary 1 *If $f_{\mathbb{Q}}$ and f_m are computable, then f is computable.*

Property 3 *Any computable real function f can be described by two (type-1) computable functions $f_{\mathbb{Q}}$ and f_m.*

Proof

Let f be a computable function and M a machine computing f. The modulus of continuity f_m is computable. This is proven in [17].

We can define $f_{\mathbb{Q}}(n,q)$ as the $(n+1)^{\text{th}}$ element of the output sequence produced by M on the constant sequence q. This function is clearly computable and gives a $2^{-(n+1)}$ approximation of $f(q)$. □

This kind of representation can be used, for example, to prove the stability of computable functions (and later of polynomial time computable real functions) by some functional operators. Here, we will prove that composition preserves computable real functions (as it preserves computable type-1 functions). Intuitively, to compute $f \circ g(x)$, we need to bound $\|g(x)\|$ to know which precision is required to compute $g(x)$.

Definition 7 (Composition) *Let us define* $(f_{\mathbb{Q}}, f_m) \circ (g_{\mathbb{Q}}, g_m) = (h_{\mathbb{Q}}, h_m)$ *by:*

$$h_{\mathbb{Q}}(p,q) = f_{\mathbb{Q}}(p+1, g_{\mathbb{Q}}(p',q))$$
$$h_m(a,p) = g_m(a, f_m(a',p)-1)$$

where $p' = f_m(\gamma(g_{\mathbb{Q}}(0,q)), p+1) - 1$ *and* $a' = max(\gamma(g_{\mathbb{Q}}(0,0))+1, g_m(a,0)+a)$

To prove that this composition over couples of such functions represents the composition of the corresponding real functions, we need the following lemma:

Lemma 1 *If* f *is represented by* f_m *and* $f_{\mathbb{Q}}$*, then* $\forall a, f(K_a) \subseteq K_{a'}$*, where* a' *is defined in definition 7.*

Proof

If $x \in K_a$, the interval bounded by 0 and x can be divided into $2^{f_m(a,0)+a}$ intervals of size less than $2^{-f_m(a,0)}$. On each interval, f cannot vary by more than $\frac{1}{2}$, by property P_1. So $\|f(x) - f(0)\| \leq 2^{f_m(a,0)+a-1}$, which gives:

$$\|f(x)\| \leq \|f(0)\| + 2^{f_m(a,0)+a-1} \leq 2^{\gamma(f_{\mathbb{Q}}(0,0))} + 2^{f_m(a,0)+a-1} \leq 2^{max(\gamma(f_{\mathbb{Q}}(0,0))+1, f_m(a,0)+a)}$$

and we have $f(x) \in K_{a'}$. □

Property 4 *If* $(f_{\mathbb{Q}}, f_m)$ *describes* f *and* $(g_{\mathbb{Q}}, g_m)$ *describes* g*, then* $(f_{\mathbb{Q}}, f_m) \circ (g_{\mathbb{Q}}, g_m)$ *describes* $f \circ g$*.*

Proof

Let us consider $(h_{\mathbb{Q}}, h_m) = (f_{\mathbb{Q}}, f_m) \circ (g_{\mathbb{Q}}, g_m)$

- Let $a, p \in \mathbb{N}$, $q \in \mathbb{R}$ and $x \in K_a$ such that $\|x - q\| \leq 2^{-h_m(a,p)}$. Then by definition of h_m, and property P_1 for g, we have $\|g(x) - g(q)\| \leq 2^{-f_m(a',p)}$. Lemma 1 ensures that $g(x) \in K_{a'}$. Then by property P_1 for f, be obtain property P_1 for $g \circ f$.
- Let $p \in \mathbb{N}$ and $q \in \mathbb{Q}$. Then

$$\|h_{\mathbb{Q}}(p,q) - f(g(q))\| = \|f_{\mathbb{Q}}(p+1, g_{\mathbb{Q}}(p',q)) - f(g(q))\|$$
$$\leq \|f_{\mathbb{Q}}(p+1, g_{\mathbb{Q}}(p',q)) - f(g_{\mathbb{Q}}(p',q))\| + \|f(g_{\mathbb{Q}}(p',q)) - f(g(q))\|$$

  by triangular inequality. The first term is bounded by $2^{-(p+2)}$ according to property P_2 for f. We have $g(q) \in K_{\gamma(g_{\mathbb{Q}}(0,q))}$ by construction of γ, and $\|g_{\mathbb{Q}}(p',q) - g(q)\| \leq 2^{-f_m(\gamma(g_{\mathbb{Q}}(0,q)),p+1)}$ by property P_2 for g and by definition of p'. Then by property P_1 for f, we obtain $\|f(p+1, g_{\mathbb{Q}}(p',q)) - f(g(q))\| \leq 2^{-(p+2)}$, then the sum of these two terms is bounded by $2^{-(p+1)}$, which gives property P_2 for h.

□

One issue is that to prove that f_m and $f_{\mathbb{Q}}$ describe a given function, we need properties P_1 and P_2, which depend on f. Here is another characterization independent from f, using the Cauchy criterion:

Property 5 *If $f_{\mathbb{Q}}$ and f_m verify the following properties then they describe a real function:*

$$P_1' : \forall a, p \in \mathbb{N}, q, q' \in K_a \cap \mathbb{Q}, \|q - q'\| \leq 2^{-f_m(a,p)} \Rightarrow \|f_{\mathbb{Q}}(p,q) - f_{\mathbb{Q}}(p,q')\| \leq 2^{-(p+2)}$$

$$P_2' : \forall p, p' \in \mathbb{N}, q \in \mathbb{Q}, \|f_{\mathbb{Q}}(p,q) - f_{\mathbb{Q}}(p',q)\| \leq 2^{-(min(p,p')+1)}$$

Proof

Assume that $f_{\mathbb{Q}}$ and f_m verify P_1' and P_2'. Then P_2' means that for all $q \in \mathbb{Q}$, $(f_{\mathbb{Q}}(p,q))_{p\in\mathbb{N}}$ is a Cauchy sequence, and so converges. This defines a function f over rational numbers. Moreover, property P_1' implies that this function is uniformly continuous over every compact of $\mathbb{Q}$. Then, f can be uniquely extended as a continuous function on $\mathbb{R}$, by density of $\mathbb{Q}$ in $\mathbb{R}$. Let us prove that $f_{\mathbb{Q}}$, f_m and f verify properties P_1 and P_2: P_2 comes naturally by taking the limit in P_2' ($p' \to \infty$). For $a, p \in \mathbb{N}, x, y \in K_a$ such that $\|x - y\| \leq 2^{-f_m(a,p)}$, there exists x_n and y_n such that $(x_n) \rightsquigarrow x$ and $(y_n) \rightsquigarrow y$ and $\forall n \in \mathbb{N}, \|x_n - y_n\| \leq 2^{-f_m(a,p)}$ (if $x \neq y$, we can take (x_n) and (y_n) between x and y). By triangular inequality, we have for all n:

$$\|f(x_n) - f(y_n)\| \leq \|f(x_n) - f_{\mathbb{Q}}(p+2, x_n)\| + \|f(y_n) - f_{\mathbb{Q}}(p+2, y_n)\| + \|f_{\mathbb{Q}}(p+2, x_n) - f_{\mathbb{Q}}(p+2, y_n)\|$$

The first two terms are bounded by $2^{-(p+3)}$ by property P_2, and the third one is bounded by $2^{-(p+2)}$ by property P_1'. The required inequality comes by taking the limit for $n \to \infty$ in $\|f(x_n) - f(y_n)\| \leq 2^{-(p+1)}$, by continuity of f. □

Remark 2. Conversely, if f is described by $f_{\mathbb{Q}}$ and f_m, it is easy to check that F_m defined by $a, p \mapsto f_m(a, p+1)$ and $F_{\mathbb{Q}}$ defined by $p, q \mapsto f_{\mathbb{Q}}(p+1, q)$ verify properties P_1' and P'_2, and these two functions also describe f. This means that the set of computable functions verifying P_1' and P_2' also characterize the set of computable real functions (and the polynomial ones characterize the set of computable polynomial functions, as defined in the following).

Note that these properties are co-semidecidable (assuming that f_m and $f_{\mathbb{Q}}$ are total), which means that we can write a program (using an implementation of f_m and $f_{\mathbb{Q}}$ terminating if and only if these properties are not verified).

2.2 Complexity

As $FPTIME$ can be defined using classical Turing machines, we can define polynomial time computable real functions on ITMs. To do so, we need to measure the size of our infinite inputs.

Definition 8 *The size of a sequence $(q_n)_{n\in\mathbb{N}}$ is the function $n \mapsto \max_{k\leq n} |q_k|$, where $|q_k|$ is the size of the binary representation of q_k.*

Definition 9 (Execution time) *A Turing machine has execution time $T : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ if on any valid input, the n^{th} element of the output sequence will be written after at most $T(n, b(n))$ steps, where $b(n)$ bounds the size of the input.*

Then, a machine has polynomial time complexity if its execution time is bounded by a polynomial.

The main difference with natural numbers is that real numbers have infinitely many representations. We will need the following lemma, expressing that any real number admits a representation with a reasonable size.

Lemma 2 *Any real number in K_a can be represented by a sequence such that the size of its n^{th} element is bounded by a polynomial in a and n.*

Proof

If $x \in K_a$, take $(\frac{\lfloor x2^n \rfloor}{2^n})_{n\in\mathbb{N}}$. Then $\lfloor x2^n \rfloor \leq 2^{a+n}$, so the encoding of $(\frac{\lfloor x2^n \rfloor}{2^n})$ is of size at most $a + 2n$ (depending on the choice of the encoding of rational numbers). □

This property is an adaptation of properties 2 and 3 for polynomial time computability (an analogous result was proven in [12] for functions defined on dyadic numbers).

Proposition 1. *A real function can be computed by a polynomial time machine if and only if it can be described by two (type-1) functions $f_{\mathbb{Q}}$ and f_m where $f_{\mathbb{Q}}$ is computable in polynomial time, and f_m is a polynomial.*

Proof

If f is described by such functions $(f_{\mathbb{Q}}, f_m)$, then we can describe the execution of a polynomial time machine computing f by looking at $@(f_{\mathbb{Q}}, f_m)$.

Conversely, let M be a polynomial time machine computing f in time P. Property 3 shows how to construct $f_{\mathbb{Q}}$ from M. This function is polynomial, by construction. We can take $f_m : a, p \mapsto P(p + 2, L(a, p + 2))$ as modulus of continuity (it is polynomial in a and p if L is the bounding polynomial of lemma 2): Let $a, p \in \mathbb{N}$, $q \in \mathbb{R}$ and $x \in K_a$ such that $\|x - q\| \leq 2^{-f_m(a,p)}$. Assume that x and q are computable. Then, from two sequences representing x and q, we can construct two sequences representing them with the same $f_m(a, p)$ first elements (e. g. any rational number between x and q) whose sizes are bounded by $L(a, p)$, according to lemma 2. Then, the execution of M on these two inputs provides the same first $(p + 2)$ output elements ($P(p + 2, L(a, p + 2))$ bounds the time needed to write these elements and then bounds the number of input elements read). This ensures that $\|f(x) - f(q)\| \leq 2^{-(p+1)}$.

Since the inequality holds for computable real numbers, by density of computable real numbers and by continuity of f, we can conclude for arbitrary input entries. □

Remark 3. It is sufficient to say that f_m is bounded by a polynomial.

As for type-1 functions, polynomial time computable functions cannot grow too fast:

Property 6 *If f is a polynomial time computable function, then there exists a polynomial P such that $\forall n, f(K_n) \subseteq K_{P(n)}$*

Proof

Lemma 1 provides the result since f_m is a polynomial. □

Remark 4. This implies that $\forall x, \|f(x)\| \leq 2^{P(\lfloor \log(\|x\|) \rfloor)}$ where P is a given polynomial. This bound can be useful to prove that a function is not polynomial time computable. It is indeed straightforward that exp is not polynomial time computable over $\mathbb{R}$.

We have characterized polynomial time computable real functions using type-1 polynomial time computable functions, which can be in turn characterized with lots of various methods, for example using functions algebras (see, for example, Bellantoni & Cook [2]), or typed lambda calculus (Leivant & Marion [13]).

3 Functions over recursively open sets

3.1 Computability

Now that we have extended the notions of computability and polynomial time complexity to functions over $\mathbb{R}$, we would like to do the same on more general domains to analyze the computability and complexity of common functions like the inverse function, the floor function, log or tan, which are not defined on $\mathbb{R}$ or on a compact set.

We will work on these particular domains:

Definition 10 *Let $D \subseteq \mathbb{R}$ be an open set. D is recursively open if $D = \mathbb{R}$ or the distance function to D^c is computable (where $D^c = \mathbb{R} \setminus D$).*

Example 1 *$\mathbb{R}\setminus\{a\}, \mathbb{R}\setminus[a, b], (a, b), \mathbb{R}\setminus E$ are recursively open sets (where a, b are computable real numbers and $E \subseteq \mathbb{N}$ is a recursive set).*

Remark 5. [12] If F is a recursively open set, then there exists $\phi : \mathbb{N} \to \mathbb{Q}$ computable such that $D = \bigcup_{n\in\mathbb{N}}(\phi(2n), \phi(2n + 1))$.

Computable functions over D can be defined analogously as in section 2.

Definition 11 *A sequence function $\tilde{f} : \mathbb{Q}^{\mathbb{N}} \to \mathbb{Q}^{\mathbb{N}}$ represents $f : D \to \mathbb{R}$ if:*

$$\forall x \in D, (q_n)_{n\in\mathbb{N}} \in \mathbb{Q}^{\mathbb{N}}, (q_n)_{n\in\mathbb{N}} \rightsquigarrow x \Rightarrow \tilde{f}((q_n)_{n\in\mathbb{N}}) \rightsquigarrow f(x)$$

Lemma 3 *The intersection of two recursively open sets is a recursively open set.*

Proof
If A and B are recursively open, then the distance functions $d(., A^c)$ and $d(., B^c)$ are computable (the case $A = \mathbb{R}$ or $B = \mathbb{R}$ is trivial). Then the function $x \mapsto \min(d(x, A^c), d(x, B^c))$ is computable, and since it is the distance function to $A^c \cup B^c = (A \cap B)^c$, $A \cap B$ is a recursively open set. □

So if f and g are respectively computable over D and D', $f + g$, $f \times g$, $min(f, g)$ and $max(f, g)$ are computable over $D \cap D'$.

It is quite simple to show that if $f(D) \subseteq D'$, then $g \circ f$ is computable over D.

As in section 2, we can describe computable functions using two type-1 functions. But $D \cap K_n$ is not necessarily a compact set, and then, a computable function does not necessarily have a modulus of continuity on it. This is why we need to define D_n as the set $\{x \mid d(x, D^c) \geq 2^{-n}\}$, i. e. the set of points of D at distance at least 2^{-n} of the boundary of D. Then, a computable function has a modulus of continuity over each compact set $D_n \cap K_n$, and we can give characterization of computable functions adapted from the one of section 2.

Definition 12 *$f : D \to \mathbb{R}$ is described by $f_{\mathbb{Q}}$ and f_m over the recursively open set D if $\forall a, p, f_m(a, p) \geq p$ and:*

$$P_1^D : \forall a, p \in \mathbb{N}, q \in \mathbb{R}, x \in K_a \cap D_a, \|x - q\| \leq 2^{-f_m(a,p)} \Rightarrow \|f(x) - f(q)\| \leq 2^{-(p+1)}$$

$$P_2^D : \forall p \in \mathbb{N}, q \in \mathbb{Q} \cap D, \|f_{\mathbb{Q}}(p, q) - f(q)\| \leq 2^{-(p+1)}$$

Property 7 *A function $f : D \to \mathbb{R}$ is computable over D if and only if it can be described over D by two (type-1) computable functions.*

Proof
To compute f on $x \in D$, compute a 2^{-n} approximation d_n of $d(x, D^c)$ until $d_n \geq 2^{-n+1}$. If such a n exists, $d(x, D^c) \geq d_n - 2^{-n} = 2^{-n}$ and $x \in D_n$. This is the case because there exists a such that $x \in D_a$, and we have: $d_{a+2} \geq d(x, D^c) - 2^{-(a+2)} = 3 \times 2^{-(a+2)} \geq 2^{-(a+1)}$. Then, we can find n' such that $x \in D_{n'} \cap K_{n'}$, compute a $2^{-f_m(n',p)}$ approximation of x and apply $f_{\mathbb{Q}}(p, .)$ on it.

This amounts to modify the γ function of section 2 to guarantee that $x \in D_{\gamma(x)} \cap K_{\gamma(x)}$ ($x \in D_{\gamma(x)}$ in section 2) and to use the same @ operator.

Conversely, the proof of section 2 still holds, except that f is only continuous on every open interval $(\phi(2n), \phi(2n+1))$, so effectively uniformly continuous on each closed interval included D_n. Then we need the condition $\forall a, p, f_m(a, p) \geq p$ to guarantee that x and y are in the same interval in property P_1^D. □

3.2 Complexity

As we made the complexity of a function on $\mathbb{R}$ depend on the size of the compact K_n on which it was calculated, we need to take into account the distance to the border of the domain. For example, the computation time of $x \mapsto 1/x$ (or log) grows when x becomes close to 0 and we need to measure this growth.

We cannot define polynomial time complexity on every recursively open set since they can be arbitrarily complex. The distance function to D^c being the most simple non trivial function on a recursively open set D, we need it to be easily computable:

Definition 13 *A polynomial open set is either $\mathbb{R}$ or a recursively open set such that the distance to its complement is computable in polynomial time (in the sense of section 2).*

Example 2 $\mathbb{R} \setminus \{a\}, \mathbb{R} \setminus [a,b], (a,b), \mathbb{R} \setminus \mathbb{N}$ *are polynomial open sets (where* a, b *are polynomial-time computable real numbers).*

Lemma 4 *If* D *is a polynomial open set, then for all* x *in* $D_a \cap K_a$*, from a representation* (q_n) *of* x*, one can compute* $a' \leq a+2$ *such that* $x \in D_{a'}$ *in time polynomial in* a*.*

Proof
The proof of property 7 describes how to get such a' in polynomial time. □

Definition 14 $f : D \to \mathbb{R}$ *has polynomial time complexity over* D *if* D *is a polynomial open set and there is a polynomial* P *and an ITM machine computing* $f(x)$ *with precision* 2^{-p} *in time* $P(a,p)$ *(in the sense of definition 9) for all* x *in* $D_a \cap K_a$*.*

Example 3 *The inverse function has polynomial time complexity over* $\mathbb{R} \setminus \{0\}$*.*

$\mathbb{R} \setminus \{0\}$ *is a polynomial open set, since the distance to* $\{0\}$ *is the absolute value, which is computable in polynomial time over* $\mathbb{R}$*. If* $\|x\| \geq 2^{-n}$ *and* $\|q - x\| < 2^{-(2n+p+2)}$*, then*

$$\|\frac{1}{x} - \frac{1}{q}\| \leq \|\frac{x-q}{xq}\| \leq \frac{2^{-(2n+p+2)}}{2^{-(2n+1)}} \leq 2^{-(p+1)}$$

To compute $\frac{1}{x}$ *with precision* 2^{-p}*, we compute* x *with precision polynomial in* n *and* p*, and then, computing the inverse of a rational number is immediate.*

This example invites us to provide a result similar to proposition 1 for functions over polynomial open sets:

Proposition 2. *A real function on a polynomial open set* D *has polynomial time complexity if and only if it can be described over* D *by two (type-1) functions* $f_{\mathbb{Q}}$ *and* f_m *where* $f_{\mathbb{Q}}$ *is computable in polynomial time over* D *(i. e. if* $q \in D_n$*,* $f_{\mathbb{Q}}(p,q)$ *can be computed in polynomial time in* p*,* n *and the size of* q*) and* f_m *is (or is bounded by) a polynomial.*

Proof
The proof of property 7 shows that in this case, the γ function and the @ operator can be computed in polynomial time. Conversely, a simple adaptation of the proof of proposition 1 shows that the $f_{\mathbb{Q}}$ function can be constructed from f and is computable in polynomial time over D, and the same modulus of continuity works. □

Remark 6. Sum, product, scalar multiplication (by a polynomially computable real number), min and max preserve polynomial time computable complexity (over the intersection of their definition domain).

This mainly comes from lemma 3 which can be adapted to prove that the intersection of two polynomial open sets is a polynomial open set (since min preserves polynomial time complexity).

Similarly to previous section, a polynomial time computable function does not grow too fast when its argument grows or becomes close to the border of the definition domain.

Property 8 *If* f *has polynomial time complexity, then there exists a polynomial* P *such that* $\forall n, f(D_n \cap K_n) \subseteq K_{P(n)}$*.*

Proof
If $x \in D_a \cap K_a$, is computable, then is has a representation (x_n) of size $n \mapsto L(a,n)$ according to lemma 2. Then, (x_{n+a+1}) is a representation of x of size $n \mapsto L(a, n+a+1)$ and $\forall n, x_{n+a+1} \in D_{a+1}$. f is polynomial time computable, so there exists a machine computing f on (x_{n+a+1}) with precision 1 in time $P(0, L(a, a+1))$ where P is a polynomial. This is polynomial in a, and the number of steps bounds the size of the output, so $f(x)$ has polynomial size with respect to a, so $\|f(x)\| \leq 2^{Q(a)}$ for some polynomial Q independent from x. □

The composition is not as easy as for functions defined everywhere since it does not always preserve polynomial time complexity:

Example 4 *$f : x \mapsto 2^{-\frac{1}{\|x\|}}$ and $g : x \mapsto \frac{1}{x}$ have both polynomial time complexity on $R \setminus \{0\}$. $f^{-1}(R^*) = R^*$, but the composition $x \mapsto 2^{\frac{1}{\|x\|}}$ is not polynomially computable on R^* for it would be in contradiction with property 8 ($g \circ f(2^{-n}) = 2^{2^n}$).*

Nevertheless, we can give some sufficient conditions to ensure safe composition.

Definition 15 *f has strong polynomial time complexity over D onto D' if f is polynomial over D and there exists a polynomial P such that $\forall n, f(D_n) \subseteq D'_{P(n)}$.*

Property 9 *If g has polynomial time complexity over D', and f has strong polynomial time complexity from D onto D', then the composition $g \circ f$ has polynomial time complexity over D.*

Proof
Since f has strong polynomial time complexity from D onto D', there exists a polynomial P such that $x \in K_n \cap D_n$, $f(x) \in D'_{P(n)}$. We compute $f(x)$ with precision $g_m(P(n), p)$ and then compute a $(p+1)$ approximation of g on the result. This takes polynomial time in n and p if f and g have polynomial time complexity over their respective domain. □

Example 5 tan *has polynomial time complexity over $D_{\tan} = \mathbb{R} \setminus \{\frac{\pi}{2} + k\pi \mid k \in \mathbb{Z}\}$. It is folklore that* sin *and* cos *are computable in polynomial time over $\mathbb{R}$.*

$$\cos((D_{\tan})_n \cap K_n) \subseteq \cos((D_{\tan})_n) = [-1,1] \setminus [\cos(\frac{\pi}{2} + 2^{-n}), \cos(\frac{\pi}{2} - 2^{-n})] =$$

$$[-1,1] \setminus [-\sin(2^{-n}), \sin(2^{-n})] \subseteq [-1,1] \setminus [-2^{-(n+1)}, 2^{-(n+1)}] \subseteq (R^*)_{n+1}$$

Then, property 9 implies that $x \mapsto \frac{1}{\cos(x)}$ has polynomial time complexity over $D_{\tan}$, and remark 6 provides that the multiplication by sin *preserves polynomial time complexity, so* tan $= \frac{\sin}{\cos}$ *has polynomial time complexity over $D_{\tan}$.*

We can find simple functions which do not verify the hypothesis of property 9 but whose composition is a polynomial-time computable function (e. g. replace g with a constant function in example 4), but strong polynomial time complexity is necessary in some sense:

Property 10 *If $f : D \to \mathbb{R}$ is computable in polynomial time over D, and if f does not have strong polynomial time complexity onto a polynomial open set D' (such that $f(D) \subseteq D'$)), then there exists a polynomial time computable function $g : D' \to \mathbb{R}$ such that the composition $g \circ f$ does not have polynomial time complexity.*

Proof
$g : x \mapsto \frac{1}{d(x, D'^c)}$ is computable in polynomial time over D' since D' is a polynomial open set (so the distance to D'^c is computable in polynomial time), the inverse function is computable in polynomial time over $\mathbb{R} \subseteq \{0\}$ (see example 3) and $x \mapsto d(x, D'^c)$ maps D'_n into $(\mathbb{R} \setminus \{0\})_n$. Since f is not strongly polynomial onto D', there exists a sequence (x_n) such that $\forall n, x_n \in D_n \cap K_n$ and $f(x_n) \notin D'_{h(n)}$ where h cannot be bounded by a polynomial. Then, $\forall n, g \circ f(x_n) \geq 2^{h(n)}$. This would be in contradiction with property 8 if $g \circ f$ were computable in polynomial time. □

To sum up, we can say that the following are equivalent (for all f with polynomial time complexity over D, and all $D' \subseteq f(D)$ polynomial open set):

- for all g with polynomial time complexity over D', $g \circ f$ has polynomial time complexity over D.
- $g \circ f$ has polynomial time complexity over D where where $g : x \mapsto \frac{1}{d(x, D^c)}$.
- f has strong polynomial time complexity over D onto D'.

The next property links functions defined over a computable interval with functions defined over $\mathbb{R}$.

Property 11 *If a and b are polynomial time computable real numbers, then there exists a polynomial-time computable bijection $h : \mathbb{R} \to (a, b)$ whose inverse is polynomially computable such that:*

– *if* $f : (a, b) \to \mathbb{R}$ *is polynomial-time computable over* (a, b)*, then* $f \circ h$ *is polynomial-time computable over* $\mathbb{R}$
– *if* $f : \mathbb{R} \to \mathbb{R}$ *is polynomial-time computable, then* $f \circ h^{-1}$ *is polynomial-time computable on* (a, b)

Proof
For the sake of simplicity, we assume $a = 0$ and $b = 1$. Since a and b are computable in polynomial time, we can prove the general case by simple translation and scaling operations, which can be done in polynomial time. We take $h(x) = \frac{x+|x|+1}{2(|x|+1)}$. h is clearly computable in polynomial time over $\mathbb{R}$. If f has polynomial time complexity over $(0, 1)$ then according to property 9, we only need to show that $h([-2^n, 2^n]) \subseteq [2^{-P(n)}, 1 - 2^{-P(n)}]$ for some polynomial P:

$$h([-2^n, 2^n]) = [\frac{1}{2(2^n + 1)}, \frac{2^{n+1} + 1}{2(2^n + 1)}] \subseteq [2^{-(n+2)}, 1 - 2^{-(n+2)}]$$

On the other hand, h^{-1} is defined by $1 - \frac{1}{2x}$ if $x < \frac{1}{2}$ and $\frac{1}{2(1-x)} - 1$ if $x \geq \frac{1}{2}$ h^{-1} behaves as the inverse function in the neighborhood of 0 and 1. A proof similar to example 3 shows that h^{-1} is also computable in polynomial time on $(0, 1)$. We also have

$$h([2^{-n}, 1 - 2^{-n}]) = [1 - 2^{n-1}, 2^{n-1} - 1] \subseteq [-2^n, 2^n]$$

and property 9 provides the polynomial complexity of $f \circ h^{-1}$ if $f : \mathbb{R} \to \mathbb{R}$ is computable in polynomial time. □

It is obvious that a (polynomially) computable function over $[0, 1]$ is also (polynomially) computable over $(0, 1)$. But conversely, a computable function over $(0, 1)$ may not have a limit in 0 or 1, and even when the limit exists, the extended function may not be computable.

Example 6 *There exists a polynomially computable function over* $(0, 1)$ *with a finite limit in* 0*, but its continuous extension to* $[0, 1)$ *is not computable.*

Proof
Pour-el and Richards [15] have shown that there exists a computable sequence (a_n) of rational numbers which converges (thus not effectively) to a non computable real number a. We can even choose this sequence computable in polynomial time (we can repeat a_n $t(n)$ times if $t(n)$ is the number of steps used to compute a_n). The piecewise linear function defined by $f(2^{-n}) = a_n$ is computable in polynomial time over $(0, 1)$ and $\lim_{x\to 0} f(x) = a$. But the extension of f onto $[0, 1)$ cannot be computable over the closed interval because a would be computable. □

The following theorem gives a simple characterization of polynomial time functions over a polynomial open set using total functions (from $\mathbb{R}^2$).

Theorem 1 *A real function* f *defined on a polynomial open set* D *has polynomial time complexity if and only if there exists* $g : \mathbb{R}^2 \to \mathbb{R}$ *with polynomial time complexity on* $\mathbb{R}^2$ *such that* $\forall x \in D, f(x) = g(x, \frac{1}{d(x,D^c)})$.

Proof
If f has polynomial time complexity over D, then we defined (informally) g by $g(x, y) = 0$ if $x \in D^c$ and if $x \in D$, $g(x, y) = f(x) \times \theta(d(x, D^c), y)$, where θ is defined as follows:

– If $|x| \geq \frac{1}{|y|+1}$ then $\theta(x, y) = 1$
– If $\frac{1}{2(|y|+1)} \leq |x| \leq \frac{1}{|y|+1}$ then $\theta(x, y) = 2|x|(|y| + 1) - 1$
– If $|x| \leq \frac{1}{2(|y|+1)}$ then $\theta(x, y) = 0$

We claim that θ is a piecewise linear function (see figure 2) and has polynomial time complexity over $\mathbb{R}^2$.

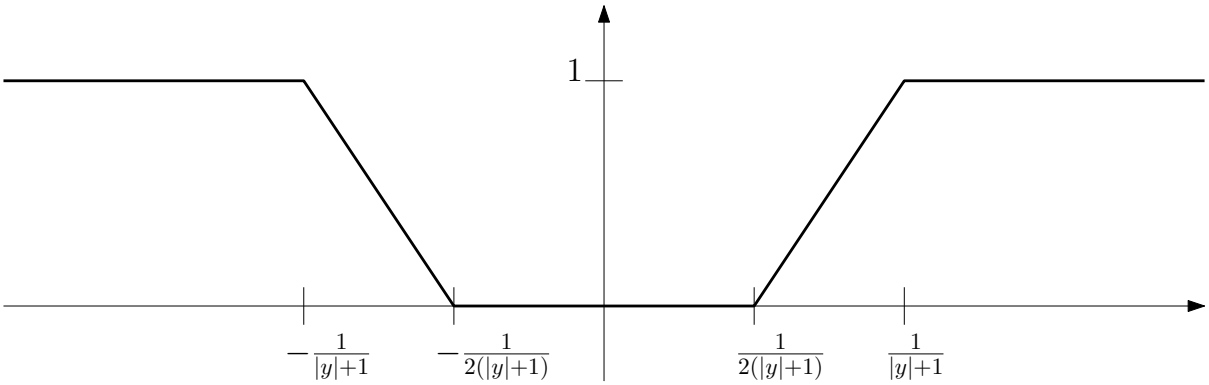

Fig. 2: The θ function for a given y.

g is computable in polynomial time on rational numbers since it only involves simple arithmetic operations and comparisons and an approximation of $f(x)$ when $d(x, D^c) \geq \frac{1}{2(|y|+1)}$. In this case, if $x \in K_a$ and $y \in K_{a'}$, $d(x, D^c) \geq \frac{1}{2(2^{a'}+1)} \geq 2^{-(a'+2)}$, thus $x \in D_{a'+2}$ and $f(x)$ with precision 2^{-p} is computable in polynomial time in p and $\max(a, a' + 2)$.

g also has a polynomial modulus of continuity since f has a polynomial modulus of continuity (in a) over the set $\{x \mid d(x, D^c) > \frac{1}{2(2^a+1)}\}$.

According to proposition 1, g has polynomial time complexity over $\mathbb{R}^2$. Now, we have that $\forall y \geq \frac{1}{d(x,D^c)}, g(x, y) = f(x)$ since in this case $d(x, D^c) \geq \frac{1}{\frac{1}{d(x,D^c)}+1} \geq \frac{1}{|y|+1}$ (this stronger property will be useful in corrolary 2).

Conversely, if g has polynomial time complexity over $\mathbb{R}^2$ and if we define f on D by $\forall x \in D, f(x) = g(x, \frac{1}{d(x,D^c)})$, then f has polynomial time complexity over D: Let x be in $K_a \cap D_a$. Then $\frac{1}{d(x,D^c)} \in K_a$ and it is computable in polynomial time in a (according to the proof of property 10) , by definition of polynomial time computability over $\mathbb{R}^2$, $g(x, \frac{1}{d(x,D^c)})$ is computable in time polynomial in a, which means that f has polynomial time complexity over D. □

The Tietze extension theorem states that any continuous function over a compact set can be extended into a continuous function over any larger compact set. Zhou [18] has proved an effective version of this theorem and here we prove a variant for polynomial time computable functions over polynomial open sets. and

Corollary 2 (Polynomial Tietze extension theorem) *f has polynomial time complexity over the polynomial open set D if and only if there exists a sequence $(f_n)_{n\in\mathbb{N}}$ of polynomial time real functions, whose complexity polynomially depends on the index n such that for all $n \in \mathbb{N}$, f_n extends $f_{|D_n}$ over $\mathbb{R}$.*

Proof

We can pose $f_n(x) = g(x, 2^n)$, where g is the function defined in theorem 1. According to the previous proof, $\forall x \in D_n, f_n(x) = f(x)$ since $\forall x \in D_n, 2^n \geq \frac{1}{d(x,D^c)}$. Since g has polynomial time complexity over $\mathbb{R}^2$ and $2^n \in K_{n+1}$, f_n has polynomial time complexity and its complexity polynomially depends on n.

Conversely, if we are provided with such functions $(f_n)_{n\in\mathbb{N}}$, we can compute $f(x) = \lim_{n\to\infty} g_n(x)$ on $x \in D_a \cap K_a$ by computing a' as defined in lemma 4 and then computing $f_{a'}(x)$. □

4 Polynomial interpretation of stream programs

In this section, we define another computation model (which is in fact equivalent to the ITM one in terms of computability), since it is more convenient for our purpose. We show that the corresponding

polynomial functions are the functions computed by programs using streams verifying some conditions. Finally, we see how this model and programs are related to polynomial time computable real functions.

4.1 Polynomial time type-2 Turing machines

We use the modified model of oracle Turing machine of Kapron and Cook [11] where oracles are functions (from $\mathbb{N}$ to $\mathbb{N}$):

Definition 16 (Oracle Turing machine) *An oracle Turing machine (also called OTM or type-2 machine in the following) $\mathcal{M}$ with k oracles and l input tapes is a Turing machine with, for each oracle, a state, one query tape and one answer tape. If $\mathcal{M}$ is used with oracles $F_1, \dots F_k : \mathbb{N} \to \mathbb{N}$, then 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 17 (Size of function) *The size of $F : \mathbb{N} \to \mathbb{N}$ is defined by: $|F|(n) = \max_{k \leq n} |F(k)|$ where $|F(k)|$ represents the size of the binary representation of $F(k)$.*

Remark 7. This notation is the same as for the size of the binary representation of an integer, but in the following, the meaning will be clear from the context.

Definition 18 (Running time of an OTM) *The weight of a step is $|F(|x|)|$ if it corresponds to a step from a query state of the oracle F on input query x and 1 otherwise. An OTM $\mathcal{M}$ has running time $T : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N} \to \mathbb{N}$ 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 weighted steps before $\mathcal{M}$ halts on these inputs is less than $T(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)$.*

Definition 19 (Second order polynomial) *A second order polynomial is a polynomial with first order and second order variables:*

$$P := c \mid X_i \mid P + P \mid P \times P \mid Y_i\langle P \rangle$$

where X_i represents a first order variable, Y_i a second order one and c a constant in $\mathbb{N}$.

The following example shows the implicit meaning of a substitution of a type-2 variable:

Example 7 *If $P(Y, X) = Y\langle Q(X) \rangle$, then if f is a function of type $\mathbb{N} \to \mathbb{N}$, then $P(f, X)$ is the polynomial $f(Q(X))$.*

In the following, $P(Y_1, \dots, Y_k, X_1, \dots, X_l)$ will implicitly denote a second order polynomial where each Y_i represents a type-2 variable, and each X_i a type-1 variable.

Definition 20 *A function $F : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ has polynomial time complexity if there exists a type 2 polynomial P such that $F(y_1, \dots, y_k, x_1, \dots x_k)$ is computed by an oracle Turing machine in time $P(|y_1|, \dots, |y_k|, |x_1|, \dots, |x_l|)$ on inputs $x_1, \dots, x_l$ and oracles $y_1, \dots, y_k$.*

Remark 8. Our model is inspired by the oracle Turing machine model used to define Basic Poly-time functionals. In this model, the output of the call of oracle F on x is not $F(|x|)$, but $F(x)$, and the size of an oracle function is $|F|(n) = \max_{|k| \leq n} |F(k)|$ (instead of $|F|(n) = \max_{k \leq n} |F(k)|$). The set of our polynomial time computable functions is then a strict subset of Basic Poly-time (proved to be equal to the BFF algebra in [11]).

The following example gives the intuition of the main difference between these two models.

Example 8 *The function $F, x \to F(|x|)$ has polynomial time complexity (bounded by $2(|x| + |F|(|x|))$: cost to copy x on the query tape and query the oracle), whereas $F, x \to F(x)$ does not (but is in BFF). Indeed, BFF functionals can access to the n^{th} value of one of their input functions in time $F(|n|)$ whereas our polynomial functionals can only access to their n^{th} element in time $F(n)$ (in this sense, F can be seen as a stream, as we will see in the following).*

The following remark shows that polynomial ITM and OTM compute the same functions in some sense.

Remark 9. If $F : ((\Sigma^*)^{\mathbb{N}})^k \to (\Sigma^*)^l \to (\Sigma^*)^{\mathbb{N}}$ is computed in polynomial time by an ITM, then $\tilde{F} : (\mathbb{N} \to \Sigma^*)^k \to (\Sigma^*)^{l+1} \to \sigma^*$ defined by

$$\tilde{F}(F_1, \dots, F_k, x_1, \dots, x_l, n) = (F((F_1(i))_{i\in\mathbb{N}}, \dots, (F_k(i))_{i\in\mathbb{N}}, x_1, \dots, x_l))_{|n|}$$

is computable in polynomial time by a type-2 Turing machine. This amounts to build an OTM which simulates the initial ITM, and between each step, writes on the simulated type-2 input tapes the successive values of $k \to F_i(k)$ (obtained by oracle calls). Instead of writing the sequence $(x_n)_{n\in\mathbb{N}}$ on an infinite tape (as in the ITM), we provide the oracle $n \to x_n$ to the OTM.

The following lemma will be useful later:

Lemma 5 *If P is a second-order polynomial, then $F_1, \dots, F_k, x_1, \dots x_l \to 2^{P(|F_1|,\dots,|F_k|,|x_1|,\dots,|x_l|)} - 1$ (i.e. the word $1 \dots 1$ of size $P(|F_1|, \dots |F_k|, |x_1|, \dots, |x_l|)$) is computable in polynomial time.*

Proof

The addition and multiplication on unary integers, and the function $x \to |x|$ are clearly computable in polynomial time. Polynomial time is also stable under composition, so we only need to prove that the size function(*i.e.* $F, n \to |F|(|n|)$ in definition 17) is computable in polynomial time. This is the case since it is a max over $|n|$ elements of length at most $|F|(|n|)$ (see lemma 8 for a functional implementation). □

4.2 Definition of the language

We define here a simple Haskell-like language where we consider streams to be infinite lists. This is a difference with Haskell, where streams are defined as finite and infinite lists, but this is not restrictive since our language also allows to define finite lists and we are only interested in showing properties of type-2 functionals. We denote by $\mathcal{F}$ the set of function symbols, $\mathcal{C}$ the set of constructors (including the stream constructor :) and $\mathcal{X}$ the set of variable names. Programs in our language are lists of definitions $\mathcal{D}$ given by the following grammar:

$$p ::= x \mid c\ p_1\ \dots\ p_n \mid p : y \text{ (Patterns)}$$
$$e ::= x \mid t\ e_1\ \dots\ e_n \text{ (Expressions)}$$
$$d ::= f\ p_1\ \dots\ p_n = e \text{ (Definitions)}$$

where $x \in \mathcal{X}, t \in \mathcal{C} \cup \mathcal{F}, c \in \mathcal{C} \setminus \{:\}$ and $f \in \mathcal{F}$. For the sake of simplicity, we only allow patterns of depth 1 for the stream constructor (*i.e.* y or $p : y$). This is not restrictive since a program with higher pattern matching depth can be easily transformed into a program of this form using some more function symbols and definitions (see remark 13). Programs can contain inductive types (denoted by `Tau` in the following), including unary integers (defined by `data Nat = 0 | Nat + 1`) and co-inductive types defined (for each inductive type `Tau`): `data [Tau] = Tau : [Tau]`. We restrict all our functions to have either type $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ or type $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{[Tau]}$ (with $k, l \geq 0$). We define lazy values by $lv ::= c\ e_1\ \dots\ e_n$ and strict values by $v ::= c\ v_1\ \dots\ v_n$. We also define the size of a closed expression: $|t\ e_1\ \dots\ e_n| = 1 + |e_1| + \dots + |e_n|$. In a definition, patterns do not share variables and all pattern matchings are exhaustive.

Derivation rules:

$$\frac{(f\ p_1\ \dots\ p_n = e) \in \mathcal{D} \quad \sigma \in \mathfrak{S} \quad \forall i, \sigma(p_i) = e_i}{f\ e_1\ \dots\ e_n \to e\{\overrightarrow{\sigma(x_i)/x_i}\}}\ (d)$$

This is the application of some definition of the function symbol f ($\mathfrak{S}$ represents the set of substitutions,*i.e.* mapping variables into expressions).

$$\frac{e_i \to e_i' \quad t \in \mathcal{F} \cup \mathcal{C} \setminus \{:\}}{t\ e_1\ \dots\ e_i\ \dots\ e_n \to t\ e_1\ \dots\ e_i'\ \dots\ e_n}\ (t)$$

This rule allows to reduce the argument of a function symbol or an expression under a constructor (different from the stream constructor).

$$\frac{e \to e'}{e : e_0 \to e' : e_0}\ (:)$$

The head of a stream can be reduces, contrary to its tail.

Notice that this derivation is not deterministic and that we can use a lazy, call-by-need strategy to mimic Haskell's semantic.

4.3 Polynomial interpretations

In the following, let positive functionals denote functions of type $(\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ with $k, l \in \mathbb{N}$.

Definition 21 (Partial order over positive functionals) *Given two positive functionals $F, G : (\mathbb{N} \to \mathbb{N})^k \to^l \mathbb{N} \to \mathbb{N}$ we define $F > G$ by:*

$$\forall F_1, \ldots, F_k : \mathbb{N} \to \mathbb{N}, \forall x_1 \ldots x_l : \mathbb{N} \setminus \{0\}, F(F_1, \ldots, F_k, x_1, \ldots, x_l) > G(F_1, \ldots, F_k, x_1, \ldots, x_l)$$

where $F_1, \ldots F_k$ are increasing functions.

Property 12 *This order is well-founded.*

Proof

The (positive) measure $F \to F(id^k, 1^l)$ is monotone (where $id : \mathbb{N} \to \mathbb{N}$ is the identity function): If $F > G$, then $F(id^k, 1^l) > G(id^k, 1^l) \geq 0$. □

Definition 22 *An interpretation $(\!|\ |\!)$ maps an expression of a program to a function. It is arbitrarily defined on the symbol functions, and the type of the interpretation is defined by the type of the symbol function by induction:*

- $(\!|\texttt{f}|\!)$ *has type* $\mathbb{N}$ *for all* `f` *of type* `Tau`
- $(\!|\texttt{f}|\!)$ *has type* $\mathbb{N} \to \mathbb{N}$ *for all* `f` *of type* `[Tau]`
- $(\!|\texttt{f}|\!)$ *has type* $T_A \to T_B$ *for all* `f` *of type* `A -> B`*, where T_A and T_B are the types of the interpretations of the symbol functions of types* `A` *and* `B`*.*

We also define the interpretation of each constructor (making a special case for the stream constructor):

- $(\!|c\ (X_1, \ldots, X_n)|\!) = X_1 + \ldots X_n + \alpha_c$ *if c is a constructor of* `Tau` *($\alpha_c \in \mathbb{N} \setminus \{0\}$. In the following we will assume, without loss of generality, that $\alpha_c = 1$).*
- $(\!|:|\!)(X, Y, Z + 1) = 1 + X + Y\langle Z \rangle$ *and* $(\!|:|\!)(X, Y, 0) = 1 + X$

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

- $(\!|x|\!) = X$ *if x is a variable of type* `Tau` *(we associate a unique type-1 variable X to each $x \in \mathcal{X}$ of type* `Tau` *).*
- $(\!|y|\!)(Z) = Y\langle Z \rangle$ *if y is a variable of type* `[Tau]` *(we associate a unique type-2 variable Y to each $y \in \mathcal{X}$ of type* `[Tau]` *).*
- $(\!|t\ e_1\ \ldots\ e_n|\!) = (\!|t|\!)((\!|e_1|\!), \ldots, (\!|e_n|\!))$ *if $t \in \mathcal{C} \cup \mathcal{F}$*

Lemma 6 *The interpretation of an expression e is a positive functional of its free variables (and of its variable Z if e has type* `[Tau]` *).*

Proof

By structural induction on the expression. This is the case for variables and for the stream constructor, the other constructors are additives, and the interpretations of function symbols are positive. □

Consequently, the interpretation of a closed expression of type `Tau` is a positive integer.

Definition 23 (Well-founded interpretation) *An interpretation of a program is well-founded if for all definition $f\ p_1 \ldots p_n = e, (\!|f\ p_1 \ldots p_n|\!) > (\!|e|\!)$. By extension we will say that such a program is well-founded.*

These are some examples of programs with well-founded polynomial interpretations.

Example 9 *The sum and product over unary integers:*

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

They admit the following interpretations: $(\!|\texttt{plus}|\!)(X_1, X_2) = 2 \times X_1 + X_2$, $(\!|\texttt{mult}|\!)(X_1, X_2) = 3 \times X_1 \times X_2$*:*

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

`s !! n`[1] *gives the* $(n+1)^{th}$ *element of the stream* `s`*:*

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

It admits the well-founded interpretation $Y, N \to Y\langle N \rangle$ *:*

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

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

In the same way, `tln` *admits the well-founded interpretation* $Y, N, Z \to Y\langle N + Z + 1 \rangle$.

Lemma 7 *If* e *and* e' *are expressions of a program with a well-founded interpretation such that* $e \to e'$*, then* $(\!|e|\!) > (\!|e'|\!)$ *.*

Proof

By structural induction on the expression. If $e \to e'$ using :

- the (d) rule, we obtain $(\!|e|\!) > (\!|e'|\!)$ using that $(\!|\ |\!)$ decreases on each definition and using that $>$ is stable by substitution ($(\!|e|\!) > (\!|e'|\!)$ is obtained by assigning interpretations, hence positive integers or increasing positive functions, to the variables of e and e').
- the (t)-rule with a function symbol f, $(\!|f\ e_1 \dots e_i \dots\ e_n|\!) > (\!|f\ e_1 \dots e'_i\ \dots\ e_n|\!)$ is obtained by definition of $>$ and since f is increasing in each variable.
- the (t)-rule with a constructor c different from :, $(\!|c\ e_1 \dots e_i \dots e_n|\!) > (\!|c\ e_1 \dots e'_i\ \dots\ e_n|\!)$ is obtained by additivity of $(\!|c|\!)$.
- the (:)-rule, if $e \to e'$, by induction hypothesis, $(\!|e|\!) > (\!|e'|\!)$, so $\forall z, (\!|e : e_0|\!)(z) = 1 + (\!|e|\!) + (\!|e_0|\!)\langle z-1 \rangle > 1 + (\!|e'|\!) + (\!|e_0|\!)\langle z-1 \rangle = (\!|e' : e_0|\!)(z)$ (this also works with $z = 0$ using the convention $(\!|e_0|\!)(-1) = 0$).

□

Corollary 3 $(\!|e|\!) - (\!|e'|\!)$ *bounds the size of the reduction of* $e \to^* e'$*.*

Corollary 4 *Every reduction chain beginning with an expression* e *of a well-founded program has its length bounded by* $(\!|e|\!)$ *.*

[1] We use the same infix notation as in Haskell.

Remark 10. This result means that to compute the n^{th} element of a stream e, we need at most $(\!|e|\!)(n)$ reduction steps (since $(\!|e\ \texttt{!!}\ n|\!) = (\!|e|\!)\langle(\!|n|\!)\rangle = (\!|e|\!)\langle n+1\rangle$ according to example 9).

The previous remark implies that every stream expression with a well-founded interpretation is productive. Productive streams are defined in the literature[9] as terms weakly normalizable to infinite lists, which is in our case equivalent to:

Definition 24 *A stream* `s` *is productive if for all* $n \in$ `Nat`, `s !! n` *evaluates to a strict value.*

In the following, we will denote by `eval` a function forcing the full evaluation (*i.e.* to a strict value) of its argument of type `Tau`.

Corollary 5 *If* f *is a function with type* $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ *of a program with a polynomial well-founded interpretation, then* `eval(f` $e_1 \dots e_n$`)` *reduces to a strict value* v *within a number of steps polynomial in* $(\!|e_1|\!), \dots, (\!|e_n|\!)$ *for all closed expressions* $e_1, \dots, e_n$.

Corollary 6 *Assume we have a (deterministic) reduction strategy. If* f *has type* $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ *in a well-founded program, then for all* $ex_1, \dots, ex_l$: `Tau` *and* $ey_1, \dots, ey_k$: `[Tau]`, `eval(f` $ey_1 \dots ey_k\ ex_1 \dots ex_l$`)` *and* `eval(f` $ey'_1 \dots ey'_k\ ex_1 \dots ex_l$`)` *reduce to the same value if for all* $n \leq N, ey_i$ `!!` n *and* ey'_i `!!` n *reduce to the same value, where* $N = (\!|\texttt{f}\ ey_1 \dots ey_k\ ex_1 \dots ex_l|\!)$.

Proof
Since pattern matchings on stream arguments have depth 1, the N^{th} element of a stream cannot be evaluated in less than N steps. `eval(f` $ey_1 \dots ey_k\ ex_1 \dots ex_l$`)` evaluates in less than N steps, so at most the N first elements of the input stream expressions can be evaluated. □

4.4 Link with polynomial time type-2 functions

Lemma 8 *Every type-2 polynomial (on unary integers) can be computed by a program with a well-founded polynomial interpretation.*

Proof
Example 9 gives an interpretation and a polynomial interpretations for the addition (`add`) and multiplication (`mult`) on unary integers (`Nat`). Then, we can define `f` computing the type-2 polynomial P by `f` $y_1 \dots y_k\ x_1 \dots x_l = e$ where e is the strict implementation of P:

- X_i is implemented by x_i
- $Y_i\langle P\rangle$ is computed by y_i `!!` P
- C where C is a constant is implemented by its unary encoding in `Nat`
- $P_1 + P_2$ is computed by `plus` $e_1\ e_2$ if e_1 and e_2 respectively compute P_1 and P_2
- $P_1 \times P_2$ is computed by `mult` $e_1\ e_2$ if e_1 and e_2 respectively compute P_1 and P_2

Since `plus` and `mult` have a polynomial interpretation, $(\!|e|\!)$ is a polynomial P_e of $y_1, \dots y_k, x_1, \dots, x_l$ and we can take $(\!|f|\!) = P_e + 1$. □

Lemma 9 *Every polynomial time type-2 function can be computed by a program with a well-founded polynomial interpretation.*

Proof

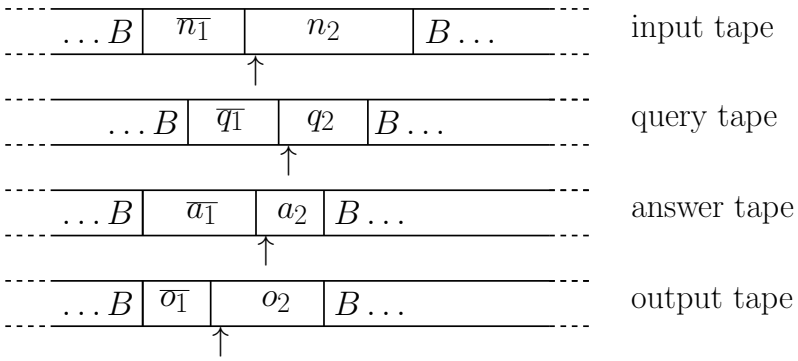

Fig. 3: Encoding of the content of the tapes of an OTM. $\overline{x}$ represents the mirror of the word x and the symbol $\uparrow$ represents the positions of the heads.

Let $f : (\mathbb{N} \to \mathbb{N})^k \to \mathbb{N}^l \to \mathbb{N}$ be a function computed by an OTM $\mathcal{M}$ in time P . For the sake of simplicity, we will assume that $k = l = 1$. The main idea is to write a function giving the output of $\mathcal{M}$ after t steps, to compute the corresponding order-2 polynomial and to apply the first function with this number of steps. We can write a program which contains unary integers, binary integers (`Bin = Nil | 0 Bin | 1 Bin`) (where `Nil` is the empty word), and a function with symbol `f0` describing the execution of $\mathcal{M}$:

```
f :: [Bin] -> Nat -> Nat -> Bin^8 -> Bin

f0 s 0 q n1 n2 q1 q2 a1 a1 o1 o2 = o2
```

Base case: if the timer is 0, then we output the content of the output tape (after its head). We also have one line of this form for each line of the transition table of $\mathcal{M}$:

```
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 `s` (of type `[Bin]`) is the stream representing the oracle, `t` (of type `Nat`) is the timer, `q` (of type `Nat`) is the index of the current state, and the other arguments represent the four tapes (*c.f.* figure 3). A tape is represented by s and $(h : t)$ if its content is s, h, t (where $\overline{s}$ is the mirror of s and $h \in \{0, 1\}$) and if its head is on the case corresponding to h. Then, we can easily emulate the moves of the heads and the shifts in the content of the tape cells. We can also emulate oracle queries: if `q` is a query state, the answer tape (`o1'` and `o2'`) will be represented by `Nil` and (`s !! q2`) after this step. 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 $(\!|f0|\!)$ since the interpretation of each of its arguments increases (from the left part to the right part of a definition) at most by a constant C (depending on the number of states), apart from $(\!|\mathtt{a2}|\!)$ which can increase by $(\!|\mathtt{s}|\!)((\!|\mathtt{q2}|\!))$ on a query state. Then, $(\!|\mathtt{f}|\!)(Y, T, X_1, \ldots, Q_2, \ldots X_n)$ can be defined by $(T + 1) \times (Y\langle Q_2\rangle + C) + X_1 + \ldots X_n$, which is strictly growing in each argument and strictly decreases on each definition. Lemma 8 shows how we can implement the polynomial P as a function p, and give it a polynomial well-founded interpretation. Finally, we pose:

```
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 (size' 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. `f1` admits a well-founded interpretation, since `max`, `size` and `maxsize` have ones:

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

□

Corollary 7 *The previous result implies that every polynomial time type-1 function (i.e. $f : \mathbb{N} \to \mathbb{N}$) can be implemented in our Haskell-like language and has a polynomial well-founded interpretation.*

The converse of lemma 9 is also true:

Lemma 10 *If a function f of type $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ admits a polynomial well-founded 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 an OTM.*

Proof

Lemma 5 shows that given some inputs and oracles, an OTM can compute $(\!|f|\!)$ applied on their sizes and get a unary integer N in polynomial time. According to corollary 6, the Haskell-like program needs at most the first N values of each oracle. Then, we can build an OTM which queries all these values (in time $\sum_{i \leq N} |f|(N)$, which is polynomial in the size of the inputs and th size of the oracles) and computes F on these finite inputs: we can convert the program computing f into a program working on finite lists (which will also have polynomial time complexity), and according to corollary 7, this type-1 program can be computed in polynomial time by a (classical) Turing machine. □

Lemmas 9 and 10 provide this equivalence:

Theorem 2 *A function f of type $\texttt{[Tau]}^k \to \texttt{Tau}^l \to \texttt{Tau}$ admits a polynomial well-founded interpretation if and only if 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 an OTM.*

Remark 11. A simple adaptation of the proofs of lemmas 9 and 10 gives a computability equivalence (a type-2 function is computable by an OTM if and only if it is computed by a program with a well-founded interpretation).

Remark 12. If we replace type-2 polynomials with functions of the form $P' := c \mid X_i \mid P' + P' \mid P' \times P' \mid Y_i\langle 2^{P'}\rangle$, we obtain a characterization of BFF. But this definition of polynomial time complexity allows an exponential (in the size of the inputs) number of derivation steps (since BFF functions can access to the n^{th} element of a type-2 input in linear time in the size of the binary representation of n).

Remark 13. We have restricted our study to programs with pattern matching of depth 1 over stream arguments. Nevertheless, we can generalize our results to arbitrary pattern matching depth by modifying our program to only have depth-one pattern matching. For example, if `f` has one argument of type `[Tau]` and the corresponding definitions do pattern matching of depth k over this argument, then we can rewrite `f` like this (for each definition of the form $\texttt{f}(p_1 : (p_2 : \ldots (p_k : t) \ldots)) = e$):

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

(where $k-1$ = `0+1+ ... +1` is the representation of $k-1$ in `Nat`, and `!!` and `tln` are defined in example 9). If `f` was interpreted by a polynomial P in the initial program, then we can interpret `f_aux` with $P_{aux}(X_1,\dots,X_k,T) = P(\sum_{1\leq i\leq k}(1+X_i)+T)$.

Indeed, $\forall Z \geq k, (e_1 : (e_2 : \dots (e_k : e)\dots))(Z) \leq \sum_{1\leq i\leq k}(1+(\!|e_i|\!)) + T(Z) (for Z \leq k$, the left term is $\sum_{1\leq i\leq Z}(1+(\!|e_i|\!)))$, and for $Z > k$ it is $\sum_{1\leq i\leq k}(1+(\!|e_i|\!)) + (\!|e|\!)(Z-k)$, which is less than $\sum_{1\leq i\leq k}(1+(\!|e_i|\!)) + (\!|e|\!)(Z)$ since e is an increasing function).

Finally, we redefine $(\!|f|\!)$ simply by $(\!|f|\!)(Y) = 1+(\!|\texttt{f\_aux}|\!)(Y\langle 1\rangle, Y\langle 2\rangle, \dots Y\langle k\rangle, Y)$, which is greater than $1+(\!|\texttt{f\_aux}|\!)(y$ `!!` $0,\dots,y$ `!!` $(k-1), Y)$. Then, a well-founded polynomial interpretation still ensures polynomial time complexity, but the interpretation does not necessarily exactly bound the number of reduction steps.

4.5 Haskell-like programs and polynomial time computable real functions

We have seen that computable real numbers can be represented as computable sequences of rational numbers, that is streams of rational numbers. Then, real functions can be seen as functions of type `[Q] -> [Q]`, where `Q` is an inductive type describing rational numbers (*e.g.* pairs of binary integers `Bin`). We will see that our notions of polynomial time complexity for stream functions and for real functions is the same in order to be able to use the results of the previous section to implement real functions with polynomial time complexity in a stream language.

This first result is straightforward.

Property 13 *If a program with a well-founded polynomial interpretation computes a real function, then this function is computable in polynomial time.*

The converse is also true:

Property 14 *Any polynomial-time computable real function (defined over $\mathbb{R}$) can be implemented by a well-founded polynomial Haskell-like program.*

Proof

According to proposition 1, such a function can be described by two type-1 poly-time functions (f_m and $f_{\mathbb{Q}}$). γ is also computable in polynomial time. Corollary 7 ensures that these three functions can be implemented by a well-founded polynomial program. Then, we can easily check that $@(f_m, f_{\mathbb{Q}})$ (defined in property 2) can be implemented using the implementation of γ, f_m and $f_{\mathbb{Q}}$ and the corresponding program admits a polynomial well-founded interpretation:

```
f_aux :: Nat -> [Q] -> [Q]
f_aux n y = (fQ n (y !! (fm (gamma (hd y)) n))) : (f_aux (n+1) y)
f :: [Q] -> [Q]
f y = f_aux 0 y
```

(where `Q` is an inductive type representing rational numbers, `fm`, `fQ` and `gamma` are the functions implementing f_m, $f_{\mathbb{Q}}$ and γ). Indeed, we can easily check that these interpretations work: $(\!|\texttt{f\_aux}|\!)(Y,N,Z) = (Z+1)\times(1+(\!|\texttt{fQ}|\!)(N+Z, Y\langle(\!|\texttt{fm}|\!)((\!|\texttt{gamma}|\!)(Y\langle 0\rangle), N+Z)|\!|))$ and $(\!|f|\!)(Y,Z) = 1+(\!|\texttt{f\_aux}|\!)(Y,1,Z)$. □

Remark 14. This result could also be proved using remark 9 which states that a computable real function can be computed by an OTM in polynomial time, and so by a well-founded polynomial program according to lemma 9, but the previous proof provides a program which only needs an implementation of f_m and $f_{\mathbb{Q}}$, and is then more constructive.

Remark 15. This result is false for real functions defined over polynomial open sets. Indeed, the gamma function is not necessarily computable in polynomial time and in this case has implementation with a well-founded polynomial interpretation. Nevertheless, according to theorem 1, it is equivalent to say that $f : D \to \mathbb{R}$ is computable in polynomial time over the polynomial open set D if and only if there exists a program with a well-founded polynomial interpretation computing g such that $\forall x \in D, f(x) = g(x, \frac{1}{d(x,D^c)})$.

5 Conclusion

We defined polynomial time complexity for real functions over quite general domains and gave a characterization using the already well known type-1 functions. It would be interesting to have a function algebra generating polynomial real functions, including some basic functions and operators like sum, product, safe composition (*i.e.* preserving polynomial time complexity, as in property 9) and probably a variation of safe recursion (as done for type-2 functions in BFF [11]) like a limit scheme (as it has been done in [7] for a different class of real functions). The set of definition domains could also be expanded (similar results can be obtained for domains D such that we only have a lower bound on the distance function $d(., D^c)$). Our last sections have shown that our definition of polynomial time computability has a real meaning when it comes to effectively programming these functions using streams, and we have provided a way to ensure polynomial time complexity, not only for real functions but also for general stream programs. The existence of a polynomial interpretation of such programs is undecidable in the general case, but we might find some heuristics or decidable subclasses, as it has been done for programs without streams (*e.g.* programs admitting an interpretation in the algebra $[\mathbb{N}, max, +]$ [1]). Other complexity classes could be characterized with a similar method. For example, space bounds have been obtained using quasi-interpretations of type-1 programs [5].

References

1. R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1):29–60, 2005.
2. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational complexity*, 2(2):97–110, 1992.
3. L. Blum, M. Shub, F. Cucker, and S. Smale. *Complexity and real computation*. Springer Verlag, 1997.
4. L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. 21(1):1–46, 1989.
5. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation: a way to control ressources. Interne, 2005.
6. O. Bournez, M. L. Campagnolo, D. S. Graça, and E. Hainry. Polynomial differential equations compute all real computable functions on computable compact intervals. *Journal of Complexity*, 23(3):317 – 335, 2007.
7. O. Bournez and E. Hainry. Elementarily computable functions over the real numbers and -sub-recursive functions. *Theoretical Computer Science*, 348(2-3):130 – 147, 2005. Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004).
8. A. Cobham. The Intrinsic Computational Difficulty of Functions. In *Logic, methodology and philosophy of science III: proceedings of the Third International Congress for Logic, Methodology and Philosophy of Science, Amsterdam 1967*, page 24. North-Holland Pub. Co., 1965.
9. J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. *Theor. Comput. Sci.*, 411(4-5):765–782, 2010.
10. J. Girard. Light linear logic. In *Logic and Computational Complexity*, pages 145–176. Springer, 1998.
11. B. Kapron and S. Cook. A new characterization of type-2 feasibility. *SIAM Journal on Computing*, 25(1):117–132, 1996.
12. K. Ko. *Complexity theory of real functions*. Birkhauser Boston Inc. Cambridge, MA, USA, 1991.
13. D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Typed Lambda Calculi and Applications*, pages 274–288, 1993.
14. R. Pechoux. *Analyse de la complexité des programmes par interprétation sémantique*. PhD thesis, Institut National Polytechnique de Lorraine - INPL, 11 2007.
15. M. B. Pour-El and I. Richards. Computability and noncomputability in classical analysis. *Transactions of the American Mathematical Society*, 275(2):539–560, 1983.
16. C. E. Shannon. Mathematical theory of the differential analyzer. *J. Math. Phys. MIT*, 20:337–354, 1941.
17. K. Weihrauch. *Computable analysis: an introduction*. Springer Verlag, 2000.
18. Q. Zhou. Computable real-valued functions on recursive open and closed subsets of Euclidean space. *Mathematical Logic Quarterly*, 42(1):379–409, 2006.