Polynomial time over the reals with parsimony

Emmanuel Hainry, Damiano Mazza, Romain Péchoux

Polynomial time over the reals with parsimony

Emmanuel Hainry[1⋆], Damiano Mazza[2⋆] and Romain Péchoux[1⋆]

[1] Project Mocqua, LORIA, CNRS, Inria, and Université de Lorraine
{hainry,pechoux}@loria.fr
[2] CNRS, UMR 7030, LIPN, Université Paris 13
damiano.mazza@lipn.univ-paris13.fr

Abstract. We provide a characterization of Ko's class of polynomial time computable functions over real numbers. This characterization holds for a stream based language using a parsimonious type discipline, a variant of propositional linear logic. We obtain a first characterization of polynomial time computations over the reals on a higher-order functional language using a linear/affine type system.

1 Introduction

Motivations. The notion of polynomial time computations over the reals has been deeply studied in the last decades, e.g. [1–4]. Several programming languages studying mathematical properties of real functions such as computable functions [5, 6], Lipschitz-functions [7], or analytical functions [8] have been introduced and studied but no programming language characterizing polynomial time over the reals has emerged. A programming language based characterization of polynomial time over the reals would be highly valuable as it would provide a programmer the opportunity to write libraries of efficient programs where any computation can be approximated in feasible time in the output precision.

This contrasts with studies on discrete domains which have led to the development by the *Implicit Computational Complexity* community of several programming languages capturing various time and space complexity classes using, among other techniques, a linear type discipline, for example see [9–12].

Characterizing complexity classes over the reals requires both convergence within a given time or space bound on inductive data (a finite approximation of the real number) and divergence on coinductive data (the infinite real number representation). If this latter requirement is not

⋆ This work was supported by ANR-14-CE25-0005 Elica: Expanding Logical Ideas for Complexity Analysis.

fulfilled then the calculus cannot be complete with respect to computability over the reals. Due to the divergence requirement, a characterization of polynomial time over the reals cannot be obtained as a straightforward extension of the results on discrete domains that enforce convergence.

Results. This paper presents a first characterization of polynomial time over the reals based on a functional higher order language. For that purpose, we consider the non-uniform parsimonious lambda-calculus that was shown to characterize the complexity classes `P/poly` and `L/poly` in [13]. Parsimony can be viewed as a logical system, a variant of multiplicative affine logic endowed with an exponential modality $!(-)$. Contrarily to any known variants of linear and affine logics, the exponential modality satisfies *Milner's law* $!A \cong A \otimes !A$. Hence it allows the programmer to encode streams (infinite lists). Non-uniformity means that we do not restrict to ultimately constant streams but allow any stream on a finite alphabet, i.e., generated by arbitrary functions with finite codomain, seen as oracles. The parsimonious calculus also enjoys the necessary property of normalizing in polynomial time on non-stream data (Theorem 1).

We characterize the class of polynomial time computable functions over the real interval [-1,1]: the n-th digit of the output can be computed in time polynomial in n (Theorem 2). Real numbers are encoded using non-uniform streams of signed binary digits (-1, 0, 1) and functions are encoded as uniform contexts that can be fed with a non-uniform real encoding (the input) under parsimony requirements.

To our knowledge, this characterization is the first linear logic based characterization of polynomial time computations over the real line. It is obtained in a very natural way by extending the indexed uniform parsimonious calculus of [14] to a non-uniform setting with streams of Church numerals as basic construct. Moreover, the semantics of the language has been improved by adopting a convention similar to [15] that uses explicit substitutions.

Related work. The parsimonious calculus can be viewed as an finite-depth/infinite-width alternative to the infinitary λ-calculus [28] where terms have finite width and infinite depth.

Studies on complexity properties of stream-based first order programming languages have already been developed using interpretation methods [16]. These only focus on soundness though. The paper [17] provides a characterization of polynomial time over the reals on first order programs using type-2 polynomial interpretations, which are however known to be untractable both at the level of inference and checking. In [18], it is shown

that algebras and coalgebras may be encoded in the light affine lambda calculus while preserving polynomial time normalization properties. The paper [19] explores applications of light logics to infinitary calculi, showing a polynomially-bounded productivity result, with no connection however to real number computation.

On function algebras, [20] and [21] provide a characterization of elementary and polynomial time computable real functions, respectively. The function coalgebra defined in [22] characterizes the complexity class of stream functions whose n-th output bit is computable in $\mathtt{L}$.

Another interesting and orthogonal line of work is the one of [23, 24] extracting certified programs for exact real number computations from constructive proofs where time complexity is not considered.

2 Preliminaries: complexity over the reals

We consider the signed binary representation of real numbers that consists in (infinite) sequences of digits in $\{1, 0, -1\}$. This representation allows numbers to be approximated from below but also from above. It is well known ([3]) that from a computability point of view, dyadic (i.e. infinite sequences of numbers of the shape $\frac{p}{2^n}; p \in \mathbb{Z}, n \in \mathbb{N}$), Cauchy and signed binary representations are equivalent. Note however that using dyadic numbers instead of a dyadic sequence gives a different (bad) notion of computability. Indeed some basic functions such as multiplication by 3 or addition are not computable if reals are restricted to dyadic numbers.

As we are mostly interested in ensuring complexity properties on the fractional part of a real number, we will restrict our analysis to real numbers in $[-1, 1]$. This restriction is standard in the exact real number computation literature [24, 6]. Our analysis can be generalized to the whole real line by just considering a pair consisting in an integer and a fractional part.

Definition 1. *Any real number $r \in [-1, 1]$ can be represented by an infinite sequence of digits in $\{-1, 0, 1\}$.*

Formally, r is represented by $\{r\} \in \{-1, 0, 1\}^{\mathbb{N}}$, noted $r \lhd \{r\}$, if

$$r = \sum_{i=1}^{\infty} \{r\}_i 2^{-i}.$$

Computing a function consists in mapping a representation of a real to a representation of its image. Based on Definition 1, a Turing machine computing a function under a signed binary encoding will work with

an infinite sequence of digits, written on a read-only input tape. The result will be written on a write-only output tape. Since the output too is infinite, the use of machines that do not halt is needed. Moreover, if we have two different representations of the same real r, then the computed outputs must represent the same real $f(r)$.

By convention, in this paper "Turing machine" will always mean a machine as just described. They are sometimes called oracle Turing Machines in the literature to exhibit the fact that the input is not a finite object but can be given by an oracle. For such a machine M and a given infinite sequence of signed digits $\{r\} \in \{-1,0,1\}^{\mathbb{N}}$, let $M(\{r\}) \in \{-1,0,1\}^{\mathbb{N}}$ be the infinite sequence of digits written (and obtained as a limit) on the output tape of the machine.

Definition 2. *A function $f : [-1,1] \to [-1,1]$ is computable if and only if there exists a Turing machine M such that $\forall r \in [-1,1]$, the following diagram commutes:*

$$\begin{array}{ccc} r & \lhd & \{r\} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle M} \\ f(r) & \lhd & M(\{r\}). \end{array}$$

The above definition implies that if a real function is computable then it must be continuous. See [3] which calls this property the *fundamental property of computable functions.*

Due to the infinite nature of computations, the complexity of real functions cannot be defined in a standard way. Ko's definition of complexity [2] associates with n the time needed to obtain precision 2^{-n}.

Definition 3. *$f : \mathbb{N} \to \mathbb{N}$ is a complexity measure for a Turing machine M if and only if $\forall n \in \mathbb{N}, \forall \{r\} \in \{-1,0,1\}^{\mathbb{N}}$, the n-th digit of $M(\{r\})$, noted $M(\{r\})_n$, is output in time less than or equal to $f(n)$.*

Definition 4. *A Turing machine M has* polynomial time complexity *if there is a polynomial $P \in \mathbb{N}[X]$ that is a complexity measure for M.*

The class of functions in $[-1,1] \to [-1,1]$ computed by machines that have polynomial time complexity is denoted $\mathrm{P}([-1,1])$.

Although the representations we mentioned (dyadic, Cauchy, signed) are equivalent from a computability point of view, they are not strictly equivalent in terms of complexity. However, the complexity class $\mathrm{P}([-1,1])$ introduced above is equal to Ko's class of polynomial time computable

functions on the dyadic representation. Indeed, from the proof of computability equivalence in [3], the translations from one representation to the other may be computed in polynomial time.

Functions computable in polynomial time can be characterized using the notion of modulus of continuity.

Definition 5 (Modulus of continuity). *Given $f : [-1,1] \to [-1,1]$, we say that $m : \mathbb{N} \to \mathbb{N}$ is a modulus of continuity for f if and only if*

$$\forall n \in \mathbb{N}, \forall r, s \in [-1,1], |r-s| < 2^{-m(n)} \implies |f(r) - f(s)| < 2^{-n}.$$

Proposition 1. *$f \in \mathrm{P}([-1,1])$ iff there exist two computable functions $m : \mathbb{N} \to \mathbb{N}$ and $\psi : [-1,1] \times \mathbb{N} \to [-1,1]$ such that*

1. *m is a polynomial modulus of continuity for f,*
2. *ψ is a polynomial time computable approximation function for f (i.e. $\forall d \in [-1,1], \forall n \in \mathbb{N}, |\psi(d,n) - f(d)| \leq 2^{-n}$).*

The above proposition is known since [2] and has been explicitly stated and formalized in [21].

Summing up, functions in $\mathrm{P}([-1,1])$ enjoy the property of being computable in polynomial time in every (dyadic) point and of having a polynomial modulus of continuity. We hence seek a stream-based programming language with a polynomially-bounded access to streams, with the aim of reproducing this property.

3 Parsimonious stream programming language

Syntax. The language under consideration will be a non-uniform version of the infinitary parsimonious lambda-calculus of [14]. Terms are defined by the following grammar:

$$(\texttt{Patterns}) \ni p ::= a \otimes b \mid !x$$
$$(\texttt{Terms}) \ni t, u ::= a \mid x_i \mid \lambda a.t \mid t\ u \mid t \otimes u \mid t[p := u] \mid !_f \overline{u} \mid t :: u \mid \mathsf{nstrm}_i$$

where: $f \in \mathbb{N} \to \mathbb{N}_k$, with $\mathbb{N}_k = \{0, \ldots, k-1\}$; $\overline{u}$ denotes a sequence of k terms $u_0, \ldots, u_{k-1}$; $a, b, c, \ldots$ range over a countable set of *affine variables*; and $x, y, z, \ldots$ range over a countable set of *exponential variables*. Exponential variables correspond to streams and are indexed by an integer $i \in \mathbb{N}$. Intuitively, x_i is the $i+1$-th element of the stream x. We always consider terms up to renaming of bound variables.

For a given function $f : \mathbb{N} \to \mathbb{N}_k$, the term $!_f \overline{u}$, called *box*, is a stream generator: $!_f(u_0, u_1, \ldots, u_{k-1})$ intuitively represents the stream $u_{f(0)} ::$

$u_{f(1)} :: u_{f(2)} :: \cdots$. The precise semantics of $!_f$ will be explained later. We use $\mathbf{u}, \mathbf{v}, \ldots$ to range over boxes. Since any f is allowed, the syntax is infinitary, except when $k = 1$, in which case f can only be the constant zero function and we simply write $!u_0$. These are called *uniform boxes*, and a term (or context) containing only uniform boxes is called *uniform*.

The explicit substitution notation $t[p := u]$ is inspired by [15] and is syntactically equivalent to the $\texttt{let}\ p := u\ \texttt{in}\ t$ construct of [14, 13]. It correspond to either a pair destructor $t[a \otimes b := u]$ or to a stream destructor $t[!x := u]$.

The *depth* $d(t)$ of a term t is the maximum number of nested boxes.

The language also includes a stream constructor :: as well as a pair constructor $\otimes$ and a family of constants nstrm_i, with $i \in \mathbb{N}$, representing streams of Church numerals (see Example 1 below) in increasing order, starting from $\underline{i}$. Let the size $|t|$ of a term t be the number of symbols in t.

Let $\diamond_1, \ldots, \diamond_n$ be special symbols called holes. A context, written $C\langle \diamond_1, \ldots, \diamond_n \rangle$ is a particular term with at most one occurence of each hole $\diamond_i$. We denote by $C\langle t_1, \ldots, t_n \rangle$ the result of substituting the term t_i to the hole $\diamond_i$ in C, an operation which may capture variables. A one-hole context, written $C\langle \diamond \rangle$ is a particular case of context with a single hole.

Example 1. Given $n \in \mathbb{N}$, the n-th *Church numeral* can be encoded as

$$\underline{n} = \lambda f.\lambda a.x_0(x_1(\ldots x_{n-1}\ a\ldots))[!x := f],$$

i.e. applications of the n first elements $x_0, \ldots, x_{n-1}$ of the stream f.

Example 2. The head function is encoded by the term

$$head = \lambda a.x_0[!x := a]$$

returning the element of index 0 in a stream a.

The tail function is encoded by the term

$$tail = \lambda a.!x_1[!x := a]$$

returning the stream of elements of the stream a starting from index 1 (i.e. $!x_1$).

Parsimony. Among terms of the language, we distinguish a family of terms called parsimonious preventing duplication of stream data and exponentiation of standard data. As we shall see later, parsimony will be entailed by the typing discipline (Lemma 1) and will ensure polynomial time normalization on non stream data (Theorem 1).

For that purpose, we first need to define a notion of slice.

Definition 6. *A* slice *of a term is obtained by removing all components but one from each box. Let $\mathcal{S}(t)$ be the set of slices of term t:*

$$\begin{aligned}
\mathcal{S}(\alpha) &:= \{\alpha\} \qquad \text{if } \alpha \in \{a, x_i\} \\
\mathcal{S}(!_f(u_0, \ldots, u_{k-1})) &:= \textstyle\bigcup_{i=0}^{k-1} \{!_f \nu_i \mid \nu_i \in \mathcal{S}(u_i)\} \\
\mathcal{S}(t_1 \otimes t_2) &:= \{\tau_1 \otimes \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(t_1\ t_2) &:= \{\tau_1\ \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(t_1 :: t_2) &:= \{\tau_1 :: \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(t_1[p := t_2]) &:= \{\tau_1[p := \tau_2] \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
\mathcal{S}(\lambda a.t) &:= \{\lambda a.\tau_1 \mid \tau_1 \in \mathcal{S}(t)\}
\end{aligned}$$

Example 3. In general, a slice does not belong to the set of (syntactically correct) terms:

$$\mathcal{S}(tail\ !_f(0,1)) = \{(\lambda a.!x_1[!x := a])\ !_f(0), (\lambda a.!x_1[!x := a])\ !_f(1)\}.$$

Definition 7. *A term t is* parsimonious *if:*

1. *affine variables appear at most once in t,*
2. *For all $s \in \mathcal{S}(t)$, two occurrences of an exponential variable in s have distinct indices,*
3. *box subterms do not contain free affine variables,*
4. *if an exponential variable appears free in t, then*
   (a) *it appears in at most one box,*
   (b) *in each slice of this box, it appears at most once,*
   (c) *if it appears in a box and outside a box, the occurrences outside a box have an index strictly smaller than those inside.*

Example 4. To illustrate Definition 7, let us see some (counter)-examples. The standard encoding of Church numerals $\lambda f.\lambda x.f\ (\ldots (f\ x)\ldots)$ breaks point 1. This is the reason why the encoding of Example 1 is used. $!_f(x_1) \otimes !_g(x_2)$ is forbidden because of point 4a. $!_f(x_0 \otimes x_1, y_0 \otimes x_1)$ is forbidden as point 4b is violated (but it respects point 2). $x_0 \otimes !_f(x_2, x_1)$ is allowed: it respects points 4b and 4c. These counter-examples are rejected as they entail either a stream duplication or a stream data duplication, whose iteration would make it possible to compute a function with an exponential modulus of continuity.

Semantics. For $k \in \mathbb{N}$, we write t^{+n} for the term obtained by replacing all free occurrences of exponential variables x_i with x_{i+n}. We write t^{++} for t^{+1} and t^{x++} to denote the same operation applied *only* to the free occurrences of the variable x. The $++$ operator is extended to sequences

by $(u_0, \ldots, u_{k-1})^{++} = (u_0^{++}, \ldots, u_{k-1}^{++})$. Also, in case all free occurrences of x have strictly positive index, we write t^{x--} for the operation which decreases indices by 1. Given a function $f : \mathbb{N} \to \mathbb{N}_k$, let f^{+i}, $i \in \mathbb{N}$, be the function in $\mathbb{N} \to \mathbb{N}_k$ defined by $\forall n \in \mathbb{N},\ f^{+i}(n) = f(n+i)$. We define *structural congruence*, denoted by $\approx$, as the smallest congruence on terms such that

$$!_f\overline{u} \ \approx \ u_{f(0)} :: !_{f^{+1}}\overline{u}^{++},$$
$$\mathsf{nstrm}_i \approx \underline{i} :: \mathsf{nstrm}_{i+1}$$

where $\underline{i}$ denotes the Church numeral encoding i described in Example 1. In particular, $\forall i > 1, !_f\overline{u} \approx u_{f(0)} :: u_{f(1)}^{++} :: \ldots :: u_{f(i-1)}^{+(i-1)} :: !_{f^{+i}}\overline{u}$ holds.

A one-hole context is *shallow* if the hole does not occur inside a box. A subclass of shallow contexts is that of *substitution* contexts, defined by:

$$[-] ::= \diamond \mid [-][p := t].$$

We write $t[-]$ instead of $[-]\langle t\rangle$. The base reduction rules of our language are:

$$\begin{array}{rll} (\lambda a.t)[-]\ u & \to_\beta & t\{u/a\}[-] \\ t[a \otimes b := (u \otimes w)[-]] & \to_\otimes & t\{u/a, w/b\}[-] \\ S\langle x_0\rangle[!x := (t :: u)[-]] & \to_{\mathtt{pop}} & S^{x--}\langle t\rangle[!x := u][-] \end{array}$$

where S is a shallow context or a term (i.e., the occurrence x_0 may actually not appear) and $t\{u/a\}$ is the standard capture-free substitution.

The operational semantics, denoted by $\to$, is defined by closing the above rules under shallow contexts and the rule stating that $t \to t'$ and $u \approx t$ implies $u \to t'$.

Example 5. Consider the term *tail* of Example 2 and let $!_f(\underline{0}, \underline{1})$ be a stream of Boolean numbers with $f : \mathbb{N} \to \{0, 1\}$ such that $f(2k) = 0$ and $f(2k+1) = 1$, for each $k \in \mathbb{N}$, i.e. $!_f(\underline{0}, \underline{1}) \approx \underline{0} :: \underline{1} :: \underline{0} :: \underline{1} :: \ldots$ As $tail = \lambda s.!x_1[!x := s]$, we have the following reduction:

$$\begin{array}{ll} tail\ !_f(\underline{0}, \underline{1}) & \\ \to_\beta\ !x_1[!x := !_f(\underline{0}, \underline{1})] & \approx (x_1 :: !x_2)[!x := \underline{0} :: !_{f^{+1}}(\underline{0}, \underline{1})] \\ \to_{\mathtt{pop}}\ (x_0 :: !x_1)[!x := !_{f^{+1}}(\underline{0}, \underline{1})] & \approx (x_0 :: !x_1)[!x := \underline{1} :: !_{f^{+2}}(\underline{0}, \underline{1})] \\ \to_{\mathtt{pop}}\ (\underline{1} :: !x_0)[!x := !_{f^{+2}}(\underline{0}, \underline{1})] & \approx (\underline{1} :: x_0 :: !x_1)[!x := \underline{0} :: !_{f^{+3}}(\underline{0}, \underline{1})] \\ \to_{\mathtt{pop}}\ \ldots & \end{array}$$

Type system. Types are defined inductively by:

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid !A \mid \forall \alpha.A.$$

α being a type variable. The type system, $\mathbf{nuPL}_\forall$, is adapted from the uniform type system of [14] and non-uniform type system of [13] and is defined in Figure 1. The notations Γ and Δ will be used for environments attributing types to variables. Judgments are of the shape $\Gamma; \Delta \vdash t : A$, Γ and Δ being two disjoint environments; meaning that term t has type A under the affine environment Δ and the exponential environment Γ. As usual, let Γ, Γ' represent the disjoint union of Γ and Γ'. Given a sequence of terms $\overline{t} = (t_1, \ldots, t_n)$ and a sequence of types $\overline{A} = (A_1, \ldots, A_n)$, let $\overline{t} : \overline{A}$ be a shorthand notation for $\forall i, t_i : A_i$. Given a sequence of exponential variables $\overline{x}$, let $\overline{x}_0$ denote the sequence obtained by indexing every variable of $\overline{x}$ by 0. The type $\mathtt{Nat}$ is a notation for the type $\forall \alpha.!(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$.

As usual a term t is *closed* if $; \vdash t : A$ can be derived, for some type A. A type A is !-free if it does not contain any occurrence of the modality !. A closed term t is of !-free type if there exists a !-free type A such that $; \vdash t : A$ can be derived.

In what follows, if A is a type with free occurrences of the type variable α and B is a type, we denote by $A[B/\alpha]$ the type obtained by replacing every occurrence of α in A with B. Let $A[]$ be a notation for $A[B/\alpha]$, for some arbitrary type B. By abuse of notation, $\mathtt{Nat}[]$ will denote the type $!(A \multimap A) \multimap A \multimap A$, for some arbitrary type A.

Definition 8 (Rank). *The rank $r(A)$ of a type A is defined by:*

$$r(\alpha) = 0 \qquad r(A \multimap B) = r(A \otimes B) = max(r(A), r(B))$$
$$r(\forall \alpha.A) = r(A) \qquad r(!A) = r(A) + 1$$

The rank $r(t)$ of a closed term t is the maximum rank of types occurring in the typing derivation for t.

The rank of a term is always well-defined as polymorphism is restricted to !-free types in rule $\forall E$, hence $r(\forall \alpha.A) = r(A[B/\alpha])$.

Example 6. The numerals of Example 1 can be given the type $\mathtt{Nat}$. We may encode unary successor and predecessor by:

$$succ = \lambda n.\lambda f.\lambda a.z_0(n\ (!z_1)\ a)[!z := f] : \mathtt{Nat} \multimap \mathtt{Nat}$$
$$pred = \lambda n.\lambda f.\lambda a.n\ ((\lambda b.b) :: (!z_0))\ a[!z := f] : \mathtt{Nat} \multimap \mathtt{Nat}$$

$$\frac{}{\Gamma;\Delta,a:A\vdash a:A}\ (\texttt{Var}) \qquad \frac{}{;\vdash \texttt{nstrm}_i:!\texttt{Nat}}\ (\texttt{Nat})$$

$$\frac{\Gamma;\Delta,a:A\vdash t:B}{\Gamma;\Delta\vdash \lambda a.t:A\multimap B}\ (\multimap\texttt{I}) \qquad \frac{\Gamma;\Delta\vdash t:A\multimap B \quad \Gamma';\Delta'\vdash u:A}{\Gamma,\Gamma';\Delta,\Delta'\vdash t\ u:B}\ (\multimap\texttt{E})$$

$$\frac{\Gamma;\Delta\vdash t:A \quad \Gamma';\Delta'\vdash u:B}{\Gamma,\Gamma';\Delta,\Delta'\vdash t\otimes u:A\otimes B}\ (\otimes\texttt{I}) \qquad \frac{\Gamma;\Delta\vdash u:A\otimes B \quad \Gamma';\Delta',a:A,b:B\vdash t:C}{\Gamma,\Gamma';\Delta,\Delta'\vdash t[a\otimes b:=u]:C}\ (\otimes\texttt{E})$$

$$\frac{\Gamma,x:A;\Delta,a:A\vdash t:B}{\Gamma,x:A;\Delta\vdash t^{x++}\{x_0/a\}:B}\ (\texttt{abs}) \qquad \frac{\Gamma;\Delta\vdash t:A \quad \Gamma';\Delta'\vdash u:!A}{\Gamma,\Gamma';\Delta,\Delta'\vdash t::u:!A}\ (\texttt{coabs})$$

$$\frac{;\overline{a}:\overline{A}\vdash \overline{u}_0:A \quad \dots \quad ;\overline{a}:\overline{A}\vdash \overline{u}_{k-1}:A}{\Gamma,\overline{x}:\overline{A};\vdash !_f\overline{u}\{\overline{x}_0/\overline{a}\}:!A}\ (!\texttt{I}) \qquad \frac{\Gamma;\Delta\vdash u:!A \quad \Gamma',x:A;\Delta'\vdash t:B}{\Gamma,\Gamma';\Delta,\Delta'\vdash t[!x:=u]:B}\ (!\texttt{E})$$

$$\frac{\Gamma;\Delta\vdash t:A \quad \alpha\notin FV(\Gamma\cup\Delta)}{\Gamma;\Delta\vdash t:\forall\alpha.A}\ (\forall\texttt{I}) \qquad \frac{\Gamma;\Delta\vdash t:\forall\alpha.A \quad B \text{ is !-free}}{\Gamma;\Delta\vdash t:A[B/\alpha]}\ (\forall\texttt{E})$$

Fig. 1. Non-uniform Parsimonious Logic Type System

Properties of **nuPL**$_\forall$. The type system enjoys some interesting properties. First, typable terms with no free exponential variables are parsimonious.

Lemma 1 (Parsimony). *If* $;\Delta\vdash t:A$ *then* t *is parsimonious.*

Second, the system enjoys subject reduction on terms with no free exponential variables:

Lemma 2 (Subject reduction). *If* $;\Delta\vdash t:A$ *and* $t\to t'$ *then* $;\Delta\vdash t':A$.

The proof of subject reduction is standard after observing that $\approx$ preserves typability, i.e. if $;\Delta\vdash t:A$ and $t\approx t'$ then $;\Delta\vdash t':A$.

Last, we can show a polynomial time normalization result on closed terms of !-free type. Let $\Lambda_{d,r}$ be the set of closed terms of !-free type of depth smaller than d and rank smaller than r.

Theorem 1. *For every* $d,r\in\mathbb{N}$*, there is a polynomial* $P_{d,r}$ *such that every term* $t\in\Lambda_{d,r}$ *normalizes in* $P_{d,r}(|t|)$ *steps.*

Proof. The proof is a rather straightforward adaptation of the proof of [13] to the considered calculus.

4 A characterization of polynomial time over the reals

Before proving the main result (Theorem 2), we introduce some preliminary notions and encodings. A function f is closed under the relation $\mathcal{R}$ if $\forall x, \forall y, x\mathcal{R}y \Rightarrow f(x) = f(y)$.

Signed bits. Consider the type $\mathtt{SB} = o \multimap o \multimap o \multimap o$ for encoding the signed binary digits $\{-1, 0, 1\}$, for some propositional variable o. Constants $\mathtt{b} \in \{-1, 0, 1\}$ can be represented by the terms $\lambda a.\lambda b.\lambda c.t$ with t equal to a, b or c depending on whether $\mathtt{b}$ is equal to 1, 0 or -1, respectively. By abuse of notation, we will use $\mathtt{b}$ to denote both the term and the constant it represents.

Real numbers. Let *toReal* be a function from $!\mathtt{SB} \to [-1, 1]$ closed under congruence $\approx$ and reduction $\to$ such that $\mathit{toReal}(\mathtt{b} :: t) = \frac{\mathtt{b}}{2} + \frac{1}{2}\mathit{toReal}(t)$.

Any real number $r \in [-1, 1]$ can be represented by a box $\mathbf{r} = !_f(-1, 0, 1)$ of type $!\mathtt{SB}$ as, by definition of *toReal*, we have

$$\mathit{toReal}(\mathbf{r}) = \sum_{i=0}^{\infty} r_{f(i)} 2^{-(i+1)}.$$

Function. A function $f : [-1, 1] \to [-1, 1]$ is *parsimoniously computable* if there is a uniform context $t_f\langle \diamond_1, \ldots, \diamond_n \rangle$ such that for each closed term t of type $!\mathtt{SB}$ we have:

$$;\vdash t_f\langle t, \ldots, t\rangle : !\mathtt{SB} \text{ and } \mathit{toReal}(t_f\langle t, \ldots, t\rangle) = f(\mathit{toReal}(t)).$$

This definition can be generalized to n-ary functions over $[-1, 1]$ by considering contexts of the shape $t_f\langle \overline{t_1}, \ldots, \overline{t_n}\rangle$ as we will see in the example computing the average of two real numbers described in Section 5.

In the above definition, the context is required to be uniform to ensure that the only real numbers we deal with are part of the input, thus preventing a non-computable oracle such as Chaitin's Ω [25] to be used.

Theorem 2. *The set of parsimoniously computable functions is exactly* $\mathrm{P}([-1, 1])$.

Proof. The proof is in 2 directions: *Soundness* and *Completeness*.

– For Soundness, we demonstrate that any parsimoniously computable function f computed by a context $t_f\langle \diamond \rangle$ is in $\mathrm{P}([-1, 1])$. For that purpose, consider the family of prefix functions computed by the terms $\mathit{get}_n : !\mathtt{SB} \multimap \mathtt{SB}^{\otimes n}$:

$$get_n = \lambda a.x_0 \otimes (\ldots \otimes x_{n-1})[!x := a],$$

where $\mathtt{SB}^{\otimes 1} = \mathtt{SB}$ and $\mathtt{SB}^{\otimes n+1} = \mathtt{SB} \otimes \mathtt{SB}^{\otimes n}$, outputting the n-th element of a stream s given as input, $n > 0$. Then, the term:

$$t_f^n\langle t\rangle = get_n\ t_f\langle t\rangle$$

can be typed by $\mathtt{SB}^{\otimes n}$.
For each n, let $toReal_n : \mathtt{SB}^{\otimes n} \to [-1, 1]$ be the function closed under reduction $\to$ and such that:

$$toReal_{n+1}(\mathsf{b} \otimes t) = \frac{\mathsf{b}}{2} + \frac{1}{2}toReal_n(t)$$
$$toReal_1(\mathsf{b}) = \frac{\mathsf{b}}{2}$$

For any $n \geq 1$ and any representation t of a real number r (i.e. $r = toReal(t)$), we have that:

$$|toReal(t_f\langle t\rangle) - toReal_n(t_f^n\langle t\rangle)| \leq 2^{-n}.$$

This is straightforward as $t_f^n\langle t\rangle$ outputs a truncation of the n first signed digits of $t_f\langle t\rangle$. Moreover $t_f^n\langle t\rangle$ is a closed term of !-free type ($\mathtt{SB}^{\otimes n}$). Consequently, it belongs to $\Lambda_{d,r}$, for some depth d and some rank r, and we can apply Theorem 1: it normalizes in $P_{d,r}(|t_f^n\langle t\rangle|)$ steps, that is in $Q(n)$ steps for some polynomial Q.

– For Completeness, we show that any function in $\mathrm{P}([-1, 1])$, computed by a Turing Machine M, is parsimoniously computable by exhibiting a uniform context with the good properties simulating M.
For that purpose, we first show how to encode iteration, duplication, and any polynomial in order to compute the polynomial time bound of the machine M.
Then, we show how to encode signed binary strings representing real numbers, how to encode operations on these strings: case, push, and pop. These operations correspond to basic tape manipulations in the Turing machine.
We also show how to encode the modulus of continuity and, finally, we encode the semantics of the Turing machine: configurations, transition function, and machinery, reusing previously introduced term for iteration.

Iteration. An iteration scheme $It(n, step, base) := n\ !(step)\ base$ corresponding to the following typing derivation, can be defined on $\mathtt{Nat}$:

$$\dfrac{\dfrac{\vdots}{;\Delta \vdash step : A \multimap A} \quad \dfrac{\vdots}{\Gamma;\Sigma \vdash base : A}}{\dfrac{\vdots}{\Gamma,\Delta';\Sigma \vdash It(n, step', base) : A}}$$

where Δ' and $step'$ are obtained from Δ and $step$, respectively, by replacing each affine variable by an exponential variable.

Duplication. We also manage to encode and type duplication on $\mathtt{Nat}$ as follows:

$$\lambda n.It(n, \lambda a.(succ\ m) \otimes (succ\ l)[m \otimes l = a], \underline{0} \otimes \underline{0}) : \mathtt{Nat}[] \multimap \mathtt{Nat} \otimes \mathtt{Nat}$$

Polynomials. Successor of Example 6 can be iterated to obtain addition of type $\mathtt{Nat}[] \multimap \mathtt{Nat} \multimap \mathtt{Nat}$. A further iteration on addition (applied to a unary integer of type $\mathtt{Nat}[]$) leads to multiplication, of type $\mathtt{Nat}[] \multimap \mathtt{Nat}[] \multimap \mathtt{Nat}$. Using addition, multiplication and duplication we may represent any polynomial with integer coefficients as a closed term of type $\mathtt{Nat}[] \multimap \mathtt{Nat}$.

The usual encodings for duplication, addition, multiplication on Church numerals are not handled as they are in need of variable duplication that is not allowed in our formalism (see Example 4).

Signed binary strings. Define the strings of signed binary numbers by $\mathtt{SB}^* = !(o \multimap o) \multimap !(o \multimap o) \multimap !(o \multimap o) \multimap o \multimap o$. The signed binary string $w = \mathtt{b}_{n-1} \dots \mathtt{b}_0 \in \{-1, 0, 1\}^n$ can be represented by the term

$$t_w = \lambda a.\lambda b.\lambda c.\lambda d.g_{n-1}(\dots g_0(d)\dots)[!z := c][!y := b][!x := a]$$

of type $\mathtt{SB}^*$, where $g_i = x_i, y_i$ or z_i depending on whether $\mathtt{b}_i = -1, 0$ or 1. In what follows, let ϵ denote the empty string of $\mathtt{SB}^*$.

Case, push and pop. We define a case construct case a of $\mathtt{b} \to t_{\mathtt{b}}$, $\mathtt{b} \in \{-1, 0, 1\}$, as syntactic sugar for $\lambda a.a\ t_1\ t_0\ t_{-1}$. Let the substitution context $[-]$ be equal to $\diamond[!x := b][!y := c][!z := d]$. We encode the

adjunction of one signed bit to a binary string by the term *push* of type $\mathtt{SB} \multimap \mathtt{SB}^* \multimap \mathtt{SB}^*$:

$$push = \text{case } a \text{ of } \mathtt{b} \to \lambda b.\lambda c.\lambda d.\lambda e.g_{\mathtt{b}}(a\ (!x_1)\ (!y_1)\ (!z_1)\ e)[-]$$

where $g_{\mathtt{b}} = x_0, y_0$ or z_0 depending on whether $\mathtt{b} = 1, 0$ or -1, respectively. In the same way we can encode the term $pop : \mathtt{SB}^* \multimap \mathtt{SB}^*$ that removes the first symbol of a String as follows:

$$\lambda a.\lambda b.\lambda c.\lambda d.\lambda e.(a\ ((\lambda f.f) :: (!x_0))\ ((\lambda f.f) :: (!y_0))((\lambda f.f) :: (!z_0))\ e)[-].$$

Modulus. We now can encode a term *modulus* of type $\mathtt{Nat}[] \multimap (\mathtt{Nat}[] \multimap \mathtt{Nat}) \multimap !\mathtt{SB} \multimap \mathtt{SB}^*$ taking the encodings of a natural number n, a modulus of continuity m, and a stream of signed digits $\{r\}$ (representing the real number r) as inputs and outputting a signed binary string corresponding to the $m(n)$ first bits of $\{r\}$, as $modulus = \lambda n.\lambda m.\lambda c.It(m\ n, \lambda a.(push\ x_0\ a), \epsilon)[!x = c]$.

Configuration and transitions. A configuration of a Turing Machine can be encoded by $\mathtt{Conf} = \mathtt{State} \otimes (\mathtt{SB}^* \otimes \mathtt{SB}^*)^k \otimes (\mathtt{SB}^* \otimes \mathtt{SB}^*)$, where:

- $\mathtt{State}$ is a type allowing to encode the finite set of states of a machine,
- each (internal) tape is encoded by a pair of signed binary string $s \otimes t : \mathtt{SB}^* \otimes \mathtt{SB}^*$. s represents the left-hand part of the tape in reverse order and t represents the right-hand part of the tape. The head points on the first digit of t.
- the last tape of type $\mathtt{SB}^* \otimes \mathtt{SB}^*$ is used to store the $m(n)$ first digits of the real number on the input tape.

The initial configuration of a machine computing a function over input $t : !\mathtt{SB}$ with modulus of continuity m can be encoded by:

$$c_0\langle t\rangle = init \otimes (\epsilon \otimes \epsilon)^k \otimes ((modulus\ n\ m\ t) \otimes \epsilon)$$

where *init* is a term encoding the initial state.

Machine. We can encode easily the transition function of the machine by a term $trans : \mathtt{Conf} \multimap \mathtt{Conf}$ using a combination of *case* constructs, and *pop* and *push* instructions on the tape. Writing a new symbol just consists in executing sequentially a pop and a push on the right part of a tape. Moving the head just consists in popping a symbol on some part of the tape and pushing on the opposite part. In the same way, we can encode a term $extract : \mathtt{Conf} \multimap \mathtt{SB}$ computing, from a final configuration, the n-th signed bit computed by the

machine. This term can also be encoded using a combination of case constructs as it suffices to read the state and the symbols to whom heads are pointing in order to compute the output.
Now consider the term $sim\langle t\rangle : \mathtt{Nat}[] \to \mathtt{SB}$ defined by:

$$sim\langle t\rangle = \lambda n.extract\ It(P\ n, trans, c_0\langle t\rangle)$$

where P encodes the polynomial time bound of the machine. By construction, $sim\langle t\rangle\ \underline{n}$ computes the term $M(\{r\})_n$, provided that $r \lhd \{r\}$ and $convert(t) = r$ both hold.

Output stream. We have demonstrated that we can compute each single signed bit of the output. It just remains to show that the output stream can be rebuilt. Consider the term $map : !(\alpha \multimap \beta) \multimap !\alpha \multimap !\beta$:

$$map := \lambda f.\lambda a.!(y_0\ x_0)[!x = a][!y = f]$$

$map\ !sim\langle t\rangle\ \mathsf{nstrm}_0$ computes the infinite stream $M(\{r\})$. Moreover the term $map\ !sim\langle\diamond\rangle\ \mathsf{nstrm}_0$ is uniform and of type $!\mathtt{SB}$. □

5 Example: Average of two real numbers

Let us now encode the average of two real numbers in $[-1, 1]$ which is equivalent to addition modulo shift but has the advantage of staying in $[-1, 1]$. Note that the difficulty is that we need to work from left to right in the stream, that is starting from the most significant bit. This uses one digit of each stream and needs a remainder between -2 and 2 coded with 2 signed digits (the remainder will be the sum of those 2 digits), see for example [24] for an Haskell implementation.

The computation of average needs three copies of the streams and is defined as $average\langle a, b\rangle := aux\langle a, b, a, b, a, b\rangle$, with $aux\langle\diamond_1, \ldots, \diamond_6\rangle :=$

$$(map\ (!\lambda n.\pi_5\ It(n, bitr\langle\diamond_5, \diamond_6\rangle, \diamond_3 \otimes \diamond_4 \otimes x_0 \otimes y_0 \otimes 0))\ \mathsf{nstrm}_0)[-],$$

$[-] := \diamond[!x := \diamond_1][!y := \diamond_2]$, and $\pi_5 := \lambda g.e[a \otimes b \otimes c \otimes d \otimes e := g]$ and using the terms It and map defined in Section 4.

The main ingredient is the term $bitr\langle a, b\rangle$ that consumes the first signed digit of each stream as well as the previous remainder and computes one digit, the next remainder and the tail of each stream. This term of type $!\mathtt{SB}^{\otimes 2} \otimes \mathtt{SB}^{\otimes 3} \multimap !\mathtt{SB}^{\otimes 2} \otimes \mathtt{SB}^{\otimes 3}$ will simply be iterated to go through the inputs and is defined by $bitr\langle\diamond_5, \diamond_6\rangle :=$

$$\lambda g.\mathrm{tail}\ \diamond_5 \otimes \mathrm{tail}\ \diamond_6 \otimes (com\ x_0\ y_0\ c\ d)[!x := a][!y := b][a \otimes b \otimes c \otimes d \otimes e := g]$$

The two first arguments are encodings of the real numbers in the input. The next two arguments represent the remainder and the last one is the computed digit. The term *com* is defined by a case analysis on digits (the case term is generalized to any number of arguments).

$$\begin{aligned} com = \text{case } x_0, y_0, c, d \text{ of } 1,1,1,1 &\to 1 \otimes 1 \otimes 1 \\ 1,1,0,1 &\to 0 \otimes 0 \otimes 1 \\ 1,1,0,0 &\to 1 \otimes 1 \otimes 0 \\ 1,1,0,-1 &\to 0 \otimes 0 \otimes 0 \\ 1,1,-1,-1 &\to -1 \otimes -1 \otimes 0 \\ 1,0,1,1 &\to 1 \otimes 0 \otimes 1 \\ \dots &\to \dots \end{aligned}$$

Indeed, the term *com* is a long sequence of all the 3^4 cases that happen for the 4 signed digits x_0, y_0, c and d (where $c + d$ represents the remainder). Instead of defining each of these cases one by one, we will give an algorithm to compute those cases:

$$com = \text{case } x_0, y_0, c, d \text{ of } \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_4 \to \mathsf{b}'_1 \otimes \mathsf{b}'_2 \otimes \mathsf{b}'_3$$

b'_3 is the result digit, computed by:

$$\begin{aligned} \mathsf{b}'_3 := \text{if } & \mathsf{b}_1 + \mathsf{b}_2 + 2\mathsf{b}_3 + 2\mathsf{b}_4 > 2 \text{ then } 1 \\ \text{elseif } & \mathsf{b}_1 + \mathsf{b}_2 + 2\mathsf{b}_3 + 2\mathsf{b}_4 < -2 \text{ then } -1 \\ \text{else } & 0 \end{aligned}$$

b'_1 and b'_2 are the remainder bits. As we use the sum of those two digits, we only give how this sum is computed, not the combination of digits that will be chosen to satisfy this constraint.

$$\mathsf{b}'_1 + \mathsf{b}'_2 = \mathsf{b}_1 + \mathsf{b}_2 + 2\mathsf{b}_3 + 2\mathsf{b}_4 - 4\mathsf{b}'_3$$

6 Remarks and future works

We have provided a first linear logic based characterization of polynomial time over the reals avoiding the use of untractable tools such as type-2 polynomials as in [17].

We now discuss some of the restrictions of our work that can be improved as future work.

– Parsimony forbids duplicating a stream, which is why we had to resort to contexts (rather than plain terms) to achieve completeness. For practical purposes, it would be preferable to add to the language an explicit controlled operator for stream duplication. Here we chose to stick to the "bare" parsimonious calculus and favor theoretical simplicity over practical usability.
– The non-polymorphic part of the type system of Fig. 1 is essentially syntax-directed: rule `abs` is the exception but it may be repeatedly applied until there are no non-linear variables at depth 0, and then one may proceed with the only applicable rule. We therefore conjecture that, with the due restrictions concerning the presence of polymorphism, type inference is efficiently decidable as in [26, 27].
– In the parsimonious calculus, it is impossible to access/manipulate the index i in x_i. The stream nstrm_i is introduced for circumventing this difficulty, which is tied to the nature of the parsimonious calculus: the syntactic tree of a term has finite depth but possibly infinite width (given by boxes). A cleaner solution, currently under investigation, would be to introduce parsimony in a calculus allowing also infinite depth, like the infinitary λ-calculus of [28], in the spirit of [19].

References

1. Müller, N.T.: Subpolynomial complexity classes of real functions and real numbers. In Kott, L., ed.: ICALP 1986. Volume 226 of LNCS. (1986) 284–293
2. Ko, K.I.: Complexity Theory of Real Functions. Birkhäuser (1991)
3. Weihrauch, K.: Computable Analysis: an Introduction. Springer (2000)
4. Kawamura, A., Cook, S.A.: Complexity theory for operators in analysis. TOCT **4**(2) (2012) 5:1–5:24
5. Brattka, V., Hertling, P.: Feasible real random access machines. J. Comp. **14**(4) (1998) 490–526
6. Ciaffaglione, A., Di Gianantonio, P.: A certified, corecursive implementation of exact real numbers. TCS **351** (2006) 39–51
7. Gianantonio, P.D., Edalat, A.: A language for differentiable functions. In: FOSSACS 2013,. (2013) 337–352
8. Ehrhard, T., Regnier, L.: The differential lambda-calculus. TCS **309**(1-3) (2003) 1–41
9. Girard, J.: Light linear logic. Inf. Comput. **143**(2) (1998) 175–204
10. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. Inf. Comput. **183**(1) (2003) 57–85
11. Gaboardi, M., Rocca, S.R.D.: A soft type assignment system for *lambda* -calculus. In: CSL 2007. (2007) 253–267
12. Baillot, P., Terui, K.: Light types for polynomial time computation in λ-calculus. Inf. Comput. **207**(1) (2009) 41–62
13. Mazza, D., Terui, K.: Parsimonious types and non-uniform computation. In: ICALP 2015, Proceedings, Part II. (2015) 350–361

14. Mazza, D.: Simple parsimonious types and logarithmic space. In: CSL 2015. (2015) 24–40
15. Accattoli, B., Dal Lago, U.: Beta reduction is invariant, indeed. In: CSL-LICS '14,. (2014) 8:1–8:10
16. Gaboardi, M., Péchoux, R.: On bounding space usage of streams using interpretation analysis. Science of Computer Programming **111** (2015) 395–425
17. Férée, H., Hainry, E., Hoyrup, M., Péchoux, R.: Characterizing polynomial time complexity of stream programs using interpretations. TCS **585** (2015) 41–54
18. Gaboardi, M., Péchoux, R.: Algebras and coalgebras in the light affine lambda calculus. In: ICFP 2015. (2015) 114–126
19. Dal Lago, U.: Infinitary lambda calculi from a linear perspective. In: LICS 2016. (2016) 447–456
20. Campagnolo, M.L.: The complexity of real recursive functions. In: UMC 2002. Volume 2509 of LNCS. (2002) 1–14
21. Bournez, O., Gomaa, W., Hainry, E.: Algebraic characterizations of complexity-theoretic classes of real functions. Int. J. Unconventional Computing **7**(5) (2011) 331–351
22. Leivant, D., Ramyaa, R. In: The Computational Contents of Ramified Corecurrence. Springer (2015) 422–435
23. Berger, U.: From coinductive proofs to exact real arithmetic: theory and applications. Logical Methods in Computer Science **7**(1) (2011)
24. Berger, U., Seisenberger, M.: Proofs, programs, processes. Theory Comput. Syst. **51**(3) (2012) 313–329
25. Chaitin, G.J.: A theory of program size formally identical to information theory. Journal of the ACM **22**(3) (1975) 329–340
26. Baillot, P.: Type inference for light affine logic via constraints on words. Theoretical Computer Science **328**(3) (2004) 289–323
27. Atassi, V., Baillot, P., Terui, K.: Verification of ptime reducibility for system F terms: Type inference in dual light affine logic. Logical Methods in Computer Science **3**(4) (2007)
28. Kennaway, J., Klop, J.W., Sleep, M.R., de Vries, F.J.: Infinitary lambda calculus. TCS **175**(1) (1997) 93–125