Higher-order interpretations for higher-order complexity

Emmanuel Hainry, Romain Péchoux

Higher-order interpretations for higher-order complexity

Emmanuel Hainry[1,2] and Romain Péchoux[1,3]

[1] Université de Lorraine, CNRS, Inria, Loria
[2] hainry@loria.fr
[3] pechoux@loria.fr

Abstract

We design an interpretation-based theory of higher-order functions that is well-suited for the complexity analysis of a standard higher- order functional language à la ML. We manage to express the interpretation of a given program in terms of a least fixpoint and we show that when restricted to functions bounded by higher-order polynomials, they characterize exactly classes of tractable functions known as Basic Feasible Functions at any order.

1 Introduction

The study of program higher-order properties remains a challenging and open issue in computer science. For example, Church-Turing's thesis does not hold at higher order and there are no complexity equivalences between computational models at higher order. As a consequence, several distinct and most of the time incomparable classes are candidate for the notion of polynomial time computation at higher order. Two of the most common definitions are the class of polynomially computable real functions by Ker-I Ko [**?**] defined for first-order functions over real numbers (that can be seen as order 2 functions over natural numbers) using machines and that was extended by Cook and Kawamura for functions at any order [**?**], and the class of Basic Feasible Functional (BFF_i) at order i by Irwin, Kapron and Royer [**?**].

As highlighted by Férée in his PhD manuscript [**?**], the classes BFF_i cannot capture some functions that can be considered to be polynomial time computable because they suffer from not accounting for the size of their higher-order arguments. However they have been demonstrated to be robust, they characterize exactly the well-known classes FPTIME and BFF at order 1 and order 2, and have already been characterized in various ways, e.g. [**?**]. An important argument in favor of the BFF_i classes is that they offer the only programming language-based alternative (BTLP, ITLP, ... - see [**?**] for an overview) for characterizing higher-order complexity. One could argue that natural programs are hard to represent in such languages unless artificial size bounds are provided. This is indeed an important issue that is solved by the present work which will separate the language, a standard higher-order language à la ML, from the size bounds, provided by static analysis.

For that purpose, we develop a theory of higher-order interpretations. These interpretations are an extension of usual (polynomial) interpretation methods used to show the termination of programs [**?**, **?**]. However this theory is a novel and uniform extension to higher-order functional programs: the definition works at any order on a simple programming language and no extra constraints are required. The only restriction lies in the reduction strategy that is enforced to be left-most outermost as our language includes operators with non-strict interpretations that could break some of the system properties if a reduction occurs under a context. We also show that these interpretations can be elegantly expressed in terms of a least fixpoint. To our knowledge, there are only two interesting lines of work that are related to our approach. In [**?**], Van De Pol introduced higher-order interpretation for showing the termination of term rewrite systems. In [**?**], Baillot and Dal Lago introduce higher-order interpretations for complexity

analysis of term rewrite systems. These two studies are restricted to variant of higher-order term rewrite systems whereas the current work applies to a functional language. While[**?**] only deals with termination properties, [**?**] is restricted to first-order complexity in contrast, our work characterizes higher-order complexity classes. Moreover [**?**] uses linear typing based extra information on functions to be applied to defunctionalized higher-order terms whereas we illustrate with an example in Section **??** that our methodology can be applied homogeneously to such a case with no need of extra or external analysis.

After defining the functional language, we present the new notion of higher-order interpretation and its properties. Next, we briefly recall the BFF_i classes and their main characterizations, including a characterization based on the BTLP programming language of [**?**]. We then demonstrate a characterization of these classes using higher-order polynomials: The soundness relies on interpretation properties: the reduction length is bounded by the interpretation of the initial term. The completeness is demonstrated by simulating a BTLP procedure, performing a compilation from procedures to terms after some program transformations. In the last section of the paper, we briefly discuss expressivity issues, the synthesis problem and possible extensions to polynomial space.

2 Functional language

Syntax. The language considered in this paper consists in a lambda calculus with constructors, primitive operators, a `case` construct for pattern matching and a `letRec` instruction for (recursive) function definitions. It can be seen as an extension of PCF to inductive data types. A term of our language is defined by the following syntax:

$$\mathtt{M},\mathtt{N} ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{op} \mid \mathtt{M}\ \mathtt{N} \mid \lambda \mathtt{x}.\mathtt{M} \mid \mathtt{letRec}\ \mathtt{f} = \mathtt{M} \mid \mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \mathtt{c}_1(\overrightarrow{\mathtt{x}_1}) \to \mathtt{M}_1 | ... | \mathtt{c}_n(\overrightarrow{\mathtt{x}_n}) \to \mathtt{M}_n$$

In the above syntax, $\mathtt{c}, \mathtt{c}_1, \cdots, \mathtt{c}_n$ are constructor symbols of fixed arity and `op` is an operator of fixed arity. Given a constructor or operator symbol b, we write $ar(b) = n$ whenever b is of arity n. Let $\mathtt{x}, \mathtt{f}$ be variables of a fixed set $\mathcal{X}$ and $\overrightarrow{\mathtt{x}_i}$ be a sequence of $ar(\mathtt{c}_i)$ variables. The free variables $FV(\mathtt{M})$ of a term `M` are defined as usual.

For simplicity, we will assume that bound variables have distinct names in order to avoid name clashes. Finally, a closed term will consists in a term `M` such that $FV(\mathtt{M}) = \emptyset$ (i.e. with no free variables).
A substitution $\{\mathtt{N}_1/\mathtt{x}_1, \cdots, \mathtt{N}_n/\mathtt{x}_n\}$ is a partial function mapping variables $\mathtt{x}_1, \cdots, \mathtt{x}_n$ to terms $\mathtt{N}_1, \cdots, \mathtt{N}_n$. As usual, the result of applying the capture-free substitution $\{\mathtt{N}_1/\mathtt{x}_1, \cdots, \mathtt{N}_n/\mathtt{x}_n\}$ to a term `M` is denoted $\mathtt{M}\{\mathtt{N}_1/\mathtt{x}_1, \cdots, \mathtt{N}_n/\mathtt{x}_n\}$ or $\mathtt{M}\{\overrightarrow{\mathtt{N}}/\overrightarrow{\mathtt{x}}\}$ when the substituting terms are clear from the context.

Semantics. The semantics of our language is described by a relation $\to_{\mathcal{S}}$ between two terms defined by $\to_{\mathcal{S}} = \to_\beta \cup \to_{\mathtt{case}} \cup \to_{\mathtt{op}} \cup \to_{\mathtt{letRec}}$

- $\to_\beta$ is the standard β-reduction defined by:

$$\lambda \mathtt{x}.\mathtt{M}\ \mathtt{N} \to_\beta \mathtt{M}\{\mathtt{N}/\mathtt{x}\},$$

- $\to_{\mathtt{case}}$ corresponds to pattern matching and is defined by:

$$\mathtt{case}\ \mathtt{c}_j(\overrightarrow{\mathtt{N}_j})\ \mathtt{of}\ \ldots \mid \mathtt{c}_j(\overrightarrow{\mathtt{x}_j}) \to \mathtt{M}_j \mid \ldots \to_{\mathtt{case}} \mathtt{M}_j\{\overrightarrow{\mathtt{N}_j}/\overrightarrow{\mathtt{x}_j}\},$$

- $\rightarrow_{\texttt{letRec}}$ is a fixpoint evaluation defined by:

$$\texttt{letRec f} = \texttt{M} \rightarrow_{\texttt{letRec}} \texttt{M}\{\texttt{letRec f} = \texttt{M}/\texttt{f}\},$$

- $\rightarrow_{\texttt{op}}$ is an operator evaluation defined by:

$$\texttt{op M}_1 \ldots \texttt{M}_n \rightarrow_{\texttt{op}} \texttt{M},$$

  provided that op is a primitive operator of arity n whose semantics $[\![\texttt{op}]\!]$, fixed by the language implementation, is a total function that satisfies $[\![\texttt{op}]\!](\texttt{M}_1, \ldots, \texttt{M}_n) = \texttt{M}$.

In the particular case, where the pattern matching on the case first argument fails (i.e. $\neg\exists j, \texttt{M} = \texttt{c}_j(\overrightarrow{\texttt{N}_j})$), we extend the relation $\rightarrow_\alpha$, $\alpha \in \{\beta, \texttt{case}, \texttt{letRec}, \texttt{op}\}$, by:

$$\text{if } \texttt{M} \rightarrow_\alpha \texttt{N} \text{ then } \texttt{case M of } \ldots \rightarrow_\alpha \texttt{case N of } \ldots.$$

In what follows, we will fix a left-most outermost evaluation strategy with respect to $\rightarrow_{\{\beta,\texttt{case},\texttt{letRec},\texttt{op}\}}$, noted $\Rightarrow$. Let $\Rightarrow^k$ be the k-fold self-composition of the relation $\Rightarrow$ with respect to such a strategy. Moreover, let $|\texttt{M} \Rightarrow^k \texttt{N}|$ be the number of reductions distinct from $\rightarrow_{\texttt{op}}$ in a given a derivation $\texttt{M} \Rightarrow^k \texttt{N}$. $|\texttt{M} \Rightarrow^k \texttt{N}| \leq k$ always holds. $[\![\texttt{M}]\!]$ is a notation for the term computed by M (if it exists), i.e. $\exists k, \texttt{M} \Rightarrow^k [\![\texttt{M}]\!]$ and $\nexists \texttt{N}, [\![\texttt{M}]\!] \Rightarrow \texttt{N}$. A (first-order) value v is defined inductively by either $\texttt{v} = \texttt{c}$, if $ar(\texttt{c}) = 0$, or $\texttt{v} = \texttt{c } \overrightarrow{\texttt{v}}$, for $ar(\texttt{c}) > 0$ values $\overrightarrow{\texttt{v}}$, otherwise.

Type system. We fix a set $\mathbf{B}$ of basic inductive types b described by their constructor set $\mathcal{C}_\texttt{b}$. For example, the type of natural numbers Nat is described by $\mathcal{C}_{\texttt{Nat}} = \{0, +1\}$. The set of simple types is defined by $\texttt{T} ::= \texttt{b} \mid \texttt{T} \longrightarrow \texttt{T}$, with $\texttt{b} \in \mathbf{B}$. As usual $\longrightarrow$ associates to the right.

In what follows, we will consider only well-typed terms. The type system assigns types to all the syntactic constructions of the language and ensures that a program does not go wrong. Notice that the typing discipline does not prevent a program from diverging. The type system is described in Figure **??** and proves judgments of the shape $\Gamma; \Delta \vdash \texttt{M} :: \texttt{T}$ meaning that the term M has type T under the variable and constructor (and operator) symbol contexts Γ and Δ respectively ; a variable (a constructor or operator symbol) context being a partial function that assigns types to variables (respectively constructors or operators). As usual, the input type and output type of constructors and operators of arity n will be restricted to basic types. Consequently, their types are of the shape $\texttt{b}_1 \longrightarrow \ldots \longrightarrow \texttt{b}_n \longrightarrow \texttt{b}$. A well-typed term will consist in a term M such that $\emptyset; \Delta \vdash \texttt{M} :: \texttt{T}$ (Consequently, it is mandatory for a term to be closed in order to be well-typed).

Definition 1 (Order)**.** *Given a term* M *of type* T*, i.e.* $\emptyset; \Delta \vdash \texttt{M} :: \texttt{T}$*, the order of* M*, denoted* $\texttt{ord}(\texttt{M})$*, is equal to the order of* T*, denoted* $\texttt{ord}(\texttt{T})$ *and defined inductively by:*

$$\begin{aligned} \texttt{ord}(\texttt{b}) &= 0, && \text{if } \texttt{b} \in \boldsymbol{B}, \\ \texttt{ord}(\texttt{T} \longrightarrow \texttt{T}') &= \max(\texttt{ord}(\texttt{T}) + 1, \texttt{ord}(\texttt{T}')) && \text{otherwise.} \end{aligned}$$

Example 1. *Consider the following term* M *that maps a function to a list given as inputs:*

$$\begin{aligned} \texttt{letRec f} = \lambda g.\lambda \texttt{x.case x of } \texttt{c}(\texttt{y}, \texttt{z}) &\rightarrow \texttt{c}(g\ \texttt{y})(\texttt{f}\ g\ \texttt{z}) \\ &\mid \texttt{nil} \rightarrow \texttt{nil} \end{aligned}$$

Suppose that [Nat] *is the base type for lists of natural numbers of constructor set* $\mathcal{C}_{[\texttt{Nat}]} = \{\texttt{nil}, \texttt{c}\}$*. The term* M *can be typed by* $(\texttt{Nat} \longrightarrow \texttt{Nat}) \longrightarrow [\texttt{Nat}] \longrightarrow [\texttt{Nat}]$*. Consequently,* $\texttt{ord}(\texttt{M}) = 2$.

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{x} :: \mathtt{T}}\ (\mathtt{Var}) \qquad \frac{\Delta(\mathtt{c}) = \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{c} :: \mathtt{T}}\ (\mathtt{Cons}) \qquad \frac{\Delta(\mathtt{op}) = \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{op} :: \mathtt{T}}\ (\mathtt{Op})$$

$$\frac{\Gamma;\Delta \vdash \mathtt{M} :: \mathtt{T}_1 \longrightarrow \mathtt{T}_2 \quad \Gamma;\Delta \vdash \mathtt{N} :: \mathtt{T}_1}{\Gamma;\Delta \vdash \mathtt{M\ N} :: \mathtt{T}_2}\ (\mathtt{App})$$

$$\frac{\Gamma, \mathtt{x} :: \mathtt{T}_1;\Delta \vdash \mathtt{M} :: \mathtt{T}_2}{\Gamma;\Delta \vdash \lambda\mathtt{x.M} :: \mathtt{T}_1 \to \mathtt{T}_2}\ (\mathtt{Abs}) \qquad \frac{\Gamma, \mathtt{f} :: \mathtt{T};\Delta \vdash \mathtt{M} :: \mathtt{T}}{\Gamma;\Delta \vdash \mathtt{letRec\ f = M\ } :: \mathtt{T}}\ (\mathtt{Let})$$

$$\frac{\Gamma;\Delta \vdash \mathtt{M} :: \mathtt{b} \qquad \Gamma;\Delta \vdash \mathtt{c}_i :: \overrightarrow{\mathtt{b}_i} \longrightarrow \mathtt{b} \quad \Gamma, \overrightarrow{\mathtt{x}_i} :: \overrightarrow{\mathtt{b}_i};\Delta \vdash \mathtt{M}_i :: \mathtt{T}\ (1 \leq i \leq n)}{\Gamma;\Delta \vdash \mathtt{case\ M\ of\ } \mathtt{c}_1(\overrightarrow{\mathtt{x}_1}) \to \mathtt{M}_1 | ... | \mathtt{c}_n(\overrightarrow{\mathtt{x}_n}) \to \mathtt{M}_n :: \mathtt{T}}\ (\mathtt{Case})$$

Figure 1: Type system

3 Interpretations

In this section, we define the interpretation tools that will allow us to control the complexity of closed terms of our language.

3.1 Interpretations of types

We briefly recall some basic definitions that are very close to the notions used in denotational semantics (See [**?**]) since, as we shall see later, our notion of interpretation also allows us to obtain fixpoints. Let $(\mathbb{N}, \leq, \sqcup, \sqcap)$ be the set of natural numbers equipped with the usual ordering $\leq$, a max operator $\sqcup$ and min operator $\sqcap$ and let $\overline{\mathbb{N}}$ be $\mathbb{N} \cup \{\top\}$, where $\forall n \in \mathbb{N},\ n \leq \top$, $n \sqcup \top = \top \sqcup n = \top$ and $n \sqcap \top = \top \sqcap n = n$. The strict order relation over natural numbers $<$ will also be used in the sequel and is extended in a somewhat unusual manner, by $\top < \top$.

The *interpretation* of a type is defined inductively by:

$$\begin{aligned} (\!|\mathtt{b}|\!) &= \overline{\mathbb{N}}, && \text{if } \mathtt{b} \text{ is a basic type,} \\ (\!|\mathtt{T} \longrightarrow \mathtt{T}'|\!) &= (\!|\mathtt{T}|\!) \longrightarrow^{\uparrow} (\!|\mathtt{T}'|\!), && \text{otherwise,} \end{aligned}$$

where $(\!|\mathtt{T}|\!) \longrightarrow^{\uparrow} (\!|\mathtt{T}'|\!)$ denotes the set of total strictly monotonic functions from $(\!|\mathtt{T}|\!)$ to $(\!|\mathtt{T}'|\!)$. A function F from the set A to the set B being strictly monotonic if for each $X, Y \in A$, $X <_A Y$ implies $F(X) <_B F(Y)$, where $<_A$ is the usual pointwise ordering induced by $<$ and defined by:

$$n <_{\overline{\mathbb{N}}} m \text{ iff } n < m$$
$$F <_{A \longrightarrow^{\uparrow} B} G \text{ iff } \forall X \in A,\ F(X) <_B G(X)$$

Example 2. *The type* $\mathtt{T} = (\mathtt{Nat} \longrightarrow \mathtt{Nat}) \longrightarrow [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]$ *of the term* $\mathtt{letRec\ f = M}$ *in Example* **??** *is interpreted by:*

$$(\!|\mathtt{T}|\!) = (\overline{\mathbb{N}} \longrightarrow^{\uparrow} \overline{\mathbb{N}}) \longrightarrow^{\uparrow} (\overline{\mathbb{N}} \longrightarrow^{\uparrow} \overline{\mathbb{N}}).$$

In what follows, given a sequence $\overrightarrow{R}$ of m terms in the interpretation domain and a sequence $\overrightarrow{\mathtt{T}}$ of k types, the notation $\overrightarrow{R} \in \overrightarrow{(\!|\mathtt{T}|\!)}$ means that both $k = m$ and $\forall i \in [1, m],\ R_i \in (\!|\mathtt{T}_i|\!)$.

3.2 Interpretations of terms

Each closed term of type $\mathtt{T}$ will be interpreted by a functional in $(\!|\mathtt{T}|\!)$. The application is denoted as usual whereas we use the notation Λ for abstraction on this function space in order to avoid confusion between terms of our calculus and objects of the interpretation domain. Variables in the interpretation domain will be denoted using upper case letters. Moreover, we will sometimes use Church typing discipline in order to highlight the type of the bound variable in a lambda abstraction.

An important distinction between the terms of our language and the objects of the interpretation domain lies in the fact that beta-reduction is considered as an equivalence relation on (closed terms of) the interpretation domain, i.e. $(\Lambda X.F)\ G = F\{G/X\}$ underlying that $(\Lambda X.F)\ G$ and $F\{G/X\}$ are distinct notations that represent the same higher-order function. The same property holds for η-reduction, i.e. $\Lambda X.(F\ X)$ and F denote the same function. In other words, closed terms of the application domain are considered as extensional objects whereas terms of the language are considered as intensional objects.

Since we are interested in complete lattices, we need to complete each type $(\!|\mathtt{T}|\!)$ by a lower bound $\bot_{(\!|\mathtt{T}|\!)}$ and an upper bound $\top_{(\!|\mathtt{T}|\!)}$ as follows:

$$\bot_{\overline{\mathbb{N}}} = 0 \qquad \top_{\overline{\mathbb{N}}} = \top$$
$$\bot_{(\!|\mathtt{T}\longrightarrow\mathtt{T}'|\!)} = \Lambda X^{(\!|\mathtt{T}|\!)}.\bot_{(\!|\mathtt{T}'|\!)} \qquad \top_{(\!|\mathtt{T}\longrightarrow\mathtt{T}'|\!)} = \Lambda X^{(\!|\mathtt{T}|\!)}.\top_{(\!|\mathtt{T}'|\!)}$$

We can show by an easy structural induction on types that for each $F \in (\!|\mathtt{T}|\!)$, $\bot_{(\!|\mathtt{T}|\!)} \leq_{(\!|\mathtt{T}|\!)} F \leq_{(\!|\mathtt{T}|\!)} \top_{(\!|\mathtt{T}|\!)}$. Notice that for each type $\mathtt{T}$ it also holds that $\top_{(\!|\mathtt{T}|\!)} <_{(\!|\mathtt{T}|\!)} \top_{(\!|\mathtt{T}|\!)}$, by an easy induction.

In the same spirit, we extend inductively the max and min operators $\sqcup$ (and $\sqcap$) over $\overline{\mathbb{N}}$ to arbitrary higher-order functions F, G of type $(\!|\mathtt{T}|\!) \longrightarrow^{\uparrow} (\!|\mathtt{T}'|\!)$ by:

$$\sqcup^{(\!|\mathtt{T}|\!)\longrightarrow^{\uparrow}(\!|\mathtt{T}'|\!)}(F,G) = \Lambda X^{(\!|\mathtt{T}|\!)}.\sqcup^{(\!|\mathtt{T}'|\!)}(F(X),G(X))$$
$$\sqcap^{(\!|\mathtt{T}|\!)\longrightarrow^{\uparrow}(\!|\mathtt{T}'|\!)}(F,G) = \Lambda X^{(\!|\mathtt{T}|\!)}.\sqcap^{(\!|\mathtt{T}'|\!)}(F(X),G(X))$$

In the following, we use the notations $\bot$, $\top$, $\leq$, $<$, $\sqcup$ and $\sqcap$ instead of $\bot_{(\!|\mathtt{T}|\!)}$, $\top_{(\!|\mathtt{T}|\!)}$, $\leq_{(\!|\mathtt{T}|\!)}$, $<_{(\!|\mathtt{T}|\!)}$, $\sqcup^{(\!|\mathtt{T}|\!)}$ and $\sqcap^{(\!|\mathtt{T}|\!)}$, respectively, when $(\!|\mathtt{T}|\!)$ is clear from the typing context.

Lemma 1. *For each type* $\mathtt{T}$, $((\!|\mathtt{T}|\!), \leq, \sqcup, \sqcap, \top, \bot)$ *is a complete lattice.*

Proof. Consider a subset S of elements in $(\!|\mathtt{T}|\!)$ and define $\sqcup S = \sqcup_{F\in S} F$. By definition, we have $F \leq \sqcup S$, for any $F \in S$. Now consider some G such that for all $F \in S$, $F \leq G$. We have for all $F \in S$, $F(X) \leq G(X)$. Consequently, $\forall X,\ S(X) = \sqcup_{F\in S} F(X) \leq G(X)$ and S is a supremum. The same holds for the infimum. □

Now we need to define a unit (or constant) cost function for any interpretation of type $\mathtt{T}$ in order to take the cost of recursive calls into account. For that purpose, let $+$ denote natural number addition extended to $\overline{\mathbb{N}}$ by $\forall n,\ \top + n = n + \top = \top$. For each type $(\!|\mathtt{T}|\!)$, we define inductively a dyadic sum function $\oplus_{(\!|\mathtt{T}|\!)}$ by:

$$X^{\overline{\mathbb{N}}} \oplus_{\overline{\mathbb{N}}} Y^{\overline{\mathbb{N}}} = X + Y$$
$$F \oplus_{(\!|\mathtt{T}\longrightarrow\mathtt{T}'|\!)} G = \Lambda X^{(\!|\mathtt{T}|\!)}.(F(X) \oplus_{(\!|\mathtt{T}'|\!)} G(X))$$

Let us also define the constant function $n_{(\!|\mathtt{T}|\!)}$, for each type $\mathtt{T}$ and each integer $n \geq 1$, by:

$$n_{\overline{\mathbb{N}}} = n$$
$$n_{(\!|\mathtt{T}\longrightarrow\mathtt{T}'|\!)} = \Lambda X^{(\!|\mathtt{T}|\!)}.n_{(\!|\mathtt{T}'|\!)}$$

- $(\!|\mathtt{f}|\!)_\rho = \rho(\mathtt{f})$, if $\mathtt{f} \in \mathcal{X}$,
- $(\!|\mathtt{c}|\!)_\rho = 1 \oplus (\Lambda X_1 . . . . \Lambda X_n . \sum_{i=1}^{n} X_i)$, if $ar(\mathtt{c}) = n$,
- $(\!|\mathtt{MN}|\!)_\rho = (\!|\mathtt{M}|\!)_\rho (\!|\mathtt{N}|\!)_\rho$,
- $(\!|\lambda \mathtt{x}.\mathtt{M}|\!)_\rho = 1 \oplus (\Lambda (\!|\mathtt{x}|\!)_\rho . (\!|\mathtt{M}|\!)_\rho)$,
- $(\!|\mathtt{case\ M\ of\ c}_1(\overrightarrow{\mathtt{x}_1}) \to \mathtt{M}_1 | ... | \mathtt{c}_n(\overrightarrow{\mathtt{x}_n}) \to \mathtt{M}_n|\!)_\rho = 1 \oplus \sqcup_{1 \leq i \leq n} \{(\!|\mathtt{M}_i|\!)_\rho \{\overrightarrow{R_i} / \overrightarrow{(\!|\mathtt{x}_i|\!)_\rho}\} \mid \forall \overrightarrow{R_i} \text{ s.t. } (\!|\mathtt{M}|\!)_\rho \geq (\!|\mathtt{c}_i|\!)_\rho \overrightarrow{R_i}\}$,
- $(\!|\mathtt{letRec\ f} = \mathtt{M}|\!)_\rho = \sqcap \{F \in (\!|\mathtt{T}|\!) \mid F \geq 1 \oplus (\Lambda (\!|\mathtt{f}|\!)_\rho . (\!|\mathtt{M}|\!)_\rho) F\}$.

Figure 2: Interpretation of a term of type $\mathtt{T}$

Once again, we will omit the type when it is unambiguous using the notation $n\oplus$ to denote the function $n_{(\!|\mathtt{T}|\!)} \oplus_{(\!|\mathtt{T}|\!)}$ when $(\!|\mathtt{T}|\!)$ is clear from the typing context. We have the following result:

Lemma 2. *For all F, $F < 1 \oplus F$.*

Proof. By induction on the strict order $<_{(\!|T|\!)}$. For the base case, we have $n <_{\overline{\mathbb{N}}} 1_{\overline{\mathbb{N}}} \oplus_{\overline{\mathbb{N}}} n = n+1$. Suppose it holds for any order $\leq k$ and take an order $k+1$ function F of type $\mathtt{T} \longrightarrow \mathtt{T}'$. For all X of type $\mathtt{T}$, we have that $FX <_{(\!|\mathtt{T}'|\!)} 1_{(\!|\mathtt{T}'|\!)} \oplus_{(\!|\mathtt{T}'|\!)} FX$, by Induction Hypothesis. Consequently $F = \Lambda X.FX <_{(\!|\mathtt{T} \to \mathtt{T}'|\!)} \Lambda X.1_{(\!|\mathtt{T}'|\!)} \oplus_{(\!|\mathtt{T}'|\!)} FX = 1_{(\!|\mathtt{T} \to \mathtt{T}'|\!)} \oplus_{(\!|\mathtt{T} \to \mathtt{T}'|\!)} F$. □

Now we are ready to define the notions of variable assignment and interpretation of a term $\mathtt{M}$:

Definition 2 (Interpretation)**.**
- *A* variable assignment*, denoted ρ, is a map associating to each $\mathtt{f} \in \mathcal{X}$ of type $\mathtt{T}$ a variable F of type $(\!|\mathtt{T}|\!)$.*
- *Given a variable assignment ρ, an* interpretation *is the extension of ρ to well-typed terms, mapping each term of type $\mathtt{T}$ to an object in $(\!|\mathtt{T}|\!)$ and defined in Figure **??**, where $(\!|\mathtt{op}|\!)_\rho$ is a sup-interpretation, i.e. a total function such that:*

$$\forall \mathtt{M}_1, \ldots, \forall \mathtt{M}_n,\ (\!|\mathtt{op\ M}_1\ \ldots\ \mathtt{M}_n|\!)_\rho \geq (\!|[\![\mathtt{op}]\!](\mathtt{M}_1, \ldots, \mathtt{M}_n)|\!)_\rho.$$

See [**?**] for more details about sup-interpretations.

As operator sup-interpretations are fixed, an interpretations should also be indexed by some mapping m assigning a sup-interpretation to each operator of the language. To simplify the formalism, we will omit this mapping in what follows.

Notice that, contrarily to other program constructs, operators may have non monotonic interpretations. This is the reason why we have fixed a left-most outermost strategy (we never reduce the operator operands) and operators are supposed to be total functions.

3.3 Existence of an interpretation

The interpretation of a term is always defined. Indeed, in Definition **??**, $(\!|\mathtt{letRec\ f} = \mathtt{M}|\!)_\rho$ is defined in terms of the least fixpoint of the function $\Lambda X^{(\!|\mathtt{T}|\!)}.1 \oplus_{(\!|\mathtt{T}|\!)} ((\Lambda (\!|\mathtt{f}|\!)_\rho . (\!|\mathtt{M}|\!)_\rho)\ X)$ and,

consequently, we obtain the following result as a direct consequence of Knaster-Tarski [**?**, **?**] Fixpoint Theorem:

Proposition 1. *Each term* $\mathtt{M}$ *of type* $\mathtt{T}$ *has an interpretation.*

Proof. By Lemma **??**, $L = (\!|\mathtt{T}|\!), \leq, \sqcup, \sqcap, \top, \bot)$ is a complete lattice. The function

$$F = \Lambda X^{(\!|\mathtt{T}|\!)}.1 \oplus_{(\!|\mathtt{T}|\!)} ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{M}|\!)_\rho)X) : L \to L$$

is monotonic. Indeed, both constructor terms and letrec terms of type $(\!|\mathtt{T}|\!)$ are interpreted over a space of monotonic functions $(\!|\mathtt{T}|\!)$. Moreover monotonicity is preserved by application, abstraction and the $\sqcap$ and $\sqcup$ operators. Applying Knaster-Tarski, we obtain that F admits a least fixpoint, which is exactly $\sqcap\{X \in (\!|\mathtt{T}|\!) \mid X \geq FX\}$. □

3.4 Intermediate lemmata

We now show intermediate lemmata by structural induction on terms

Lemma 3. *For all* $\mathtt{M}, \mathtt{N}, \mathtt{x}$ *such that* $\mathtt{x} :: \mathtt{T}; \Delta \vdash \mathtt{M} :: \mathtt{T}'$, $\emptyset; \Delta \vdash \mathtt{N} :: \mathtt{T}$, *we have* $(\!|\mathtt{M}|\!)_\rho\{(\!|\mathtt{N}|\!)_\rho/(\!|\mathtt{x}|\!)_\rho\} = (\!|\mathtt{M}\{\mathtt{N}/\mathtt{x}\}|\!)_\rho$.

Lemma 4. *For all* $\mathtt{M}, \mathtt{N}, \mathtt{x}$ *such that* $\mathtt{x} :: \mathtt{T}; \Delta \vdash \mathtt{M} :: \mathtt{T}'$, $\emptyset; \Delta \vdash \mathtt{N} :: \mathtt{T}$, *we have* $(\!|\lambda\mathtt{x}.\mathtt{M}\ \mathtt{N}|\!)_\rho > (\!|\mathtt{M}\{\mathtt{N}/\mathtt{x}\}|\!)_\rho$.

Proof.

$$\begin{aligned}
(\!|\lambda\mathtt{x}.\mathtt{M}\ \mathtt{N}|\!)_\rho &= (\!|\lambda\mathtt{x}.\mathtt{M}|\!)_\rho(\!|\mathtt{N}|\!)_\rho && \text{By Definition ??}\\
&= \Lambda(\!|\mathtt{x}|\!)_\rho.1 \oplus (\!|\mathtt{M}|\!)_\rho(\!|\mathtt{N}|\!)_\rho && \text{By Definition ??}\\
&= 1 \oplus (\!|\mathtt{M}|\!)_\rho\{(\!|\mathtt{N}|\!)_\rho/(\!|\mathtt{x}|\!)_\rho\} && \text{By definition of } =\\
&= 1 \oplus (\!|\mathtt{M}\{\mathtt{N}/\mathtt{x}\}|\!)_\rho && \text{By Lemma ??}\\
&> (\!|\mathtt{M}\{\mathtt{N}/\mathtt{x}\}|\!)_\rho && \text{By Lemma ??}
\end{aligned}$$

and so the conclusion. □

Lemma 5. *For all* $\mathtt{M}$, *we have: if* $\mathtt{M} \Rightarrow \mathtt{N}$ *then* $(\!|\mathtt{M}|\!)_\rho \geq (\!|\mathtt{N}|\!)_\rho$. *Moreover if* $|\mathtt{M} \Rightarrow \mathtt{N}| = 1$ *then* $(\!|\mathtt{M}|\!)_\rho > (\!|\mathtt{N}|\!)_\rho$.

Proof. If $|\mathtt{M} \Rightarrow \mathtt{N}| = 0$ then $\mathtt{M} = \mathtt{op}\ \mathtt{M}_1\ \ldots\ \mathtt{M}_n \to_{\mathtt{op}} [\![\mathtt{op}]\!](\mathtt{M}_1, \ldots, \mathtt{M}_n) = \mathtt{N}$, for some operator $\mathtt{op}$ and terms $\mathtt{M}_1, \ldots, \mathtt{M}_n$ and consequently, by Definition of interpretations we have $(\!|\mathtt{op}\ \mathtt{M}_1\ \ldots\ \mathtt{M}_n|\!)_\rho \geq (\!|[\![\mathtt{op}]\!](\mathtt{M}_1, \ldots, \mathtt{M}_n)|\!)_\rho$.

If $|\mathtt{M} \Rightarrow \mathtt{N}| = 1$ then the reduction does not involve a $\to_{\mathtt{op}}$ (i.e. an operator evaluation). By Lemma **??**, in the case of a β-reduction and, by induction, Lemmata **??** and **??** and Definition **??** for the other cases. e.g. For a letrec reduction, we have: if $\mathtt{M} = \mathtt{letRec}\ \mathtt{f} = \mathtt{M}' \to_{\mathtt{letRec}} \mathtt{M}'\{\mathtt{M}/\mathtt{f}\} = \mathtt{N}$ then:

$$\begin{aligned}
(\!|\mathtt{M}|\!)_\rho &= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)F\}\\
&\geq \sqcap\{1 \oplus \Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho F \mid F \geq 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)F\}\\
&\geq 1 \oplus \Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho \sqcap \{F \mid F \geq 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)F\}\\
&= 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)(\!|\mathtt{M}|\!)_\rho\\
&= 1 \oplus (\!|\mathtt{N}|\!)_\rho\{(\!|\mathtt{M}|\!)_\rho/(\!|\mathtt{f}|\!)_\rho\}\\
&= 1 \oplus (\!|\mathtt{N}\{\mathtt{M}/\mathtt{f}\}|\!)_\rho, \quad \text{by Lemma ??}\\
&> (\!|\mathtt{N}\{\mathtt{M}/\mathtt{f}\}|\!)_\rho, \quad \text{by Lemma ??}
\end{aligned}$$

$$
\begin{aligned}
&(\!|\mathtt{M}|\!)_\rho \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\lambda \mathtt{x.case\ x\ of\ } +\mathtt{1(y)} \to +\mathtt{1}(+\mathtt{1(f\ y)})\mid 0 \to 0|\!)_\rho)F\} \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 2 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(\!|\mathtt{case\ x\ of\ } +\mathtt{1(y)} \to +\mathtt{1}(+\mathtt{1(f\ y)})\mid 0 \to 0|\!)_\rho)F)\} \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 3 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(\sqcup_{(\!|\mathtt{x}|\!)_\rho \geq (\!|+\mathtt{1(y)}|\!)_\rho}(\!|+\mathtt{1}(+\mathtt{1(f\ y)})|\!)_\rho) \sqcup (\sqcup_{(\!|\mathtt{x}|\!)_\rho \geq (\!|\mathtt{0}|\!)_\rho}(\!|\mathtt{0}|\!)_\rho))F)\} \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 3 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(\sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho} 2 \oplus ((\!|\mathtt{f}|\!)_\rho\ (\!|\mathtt{y}|\!)_\rho)) \sqcup (\sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1} 1))F)\} \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 3 \oplus (\Lambda(\!|\mathtt{x}|\!)_\rho.(\sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho} 2 \oplus (F\ (\!|\mathtt{y}|\!)_\rho)) \sqcup (1))\} \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 3 \oplus (\Lambda(\!|\mathtt{x}|\!)_\rho.(2 \oplus (F\ ((\!|\mathtt{x}|\!)_\rho - 1))) \sqcup (1))\}, \quad (\!|\mathtt{x}|\!)_\rho - 1 \geq 0 \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq \Lambda X.(5 \oplus (F\ (X-1))) \sqcup (4)\} \\
&= \Lambda X.5X + 4
\end{aligned}
$$

Figure 3: Example of interpretation

The first inequality holds since we are only considering higher-order functions F satisfying $F \geq 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)F$. The second inequality holds because $\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho$ is a monotonic function (as the interpretation domain only consists of such functions). □

Corollary 1. *For all terms,* $\mathtt{M}$, $\overrightarrow{\mathtt{N}}$, *such that* $\emptyset; \Delta \vdash \mathtt{M}\ \overrightarrow{\mathtt{N}} :: \mathtt{T}$, *if* $\mathtt{M}\ \overrightarrow{\mathtt{N}} \Rightarrow^k \mathtt{M'}$ *then* $(\!|\mathtt{M}|\!)_\rho(\!|\overrightarrow{\mathtt{N}}|\!)_\rho \geq |\mathtt{M}\ \overrightarrow{\mathtt{N}} \Rightarrow^k \mathtt{M'}| \oplus (\!|\mathtt{M'}|\!)_\rho$.

As basic operators can be considered as constant time computable objects the following Corollary also holds:

Corollary 2. *For all terms,* $\mathtt{M}$, $\overrightarrow{\mathtt{N}}$, *such that* $\emptyset; \Delta \vdash \mathtt{M}\ \overrightarrow{\mathtt{N}} :: \mathtt{b}$, *with* $\mathtt{b} \in \boldsymbol{B}$, *if* $(\!|\mathtt{M}\ \overrightarrow{\mathtt{N}}|\!)_\rho \neq \top$ *then* $\mathtt{M}\ \overrightarrow{\mathtt{N}}$ *terminates in a number of reduction steps in* $O((\!|\mathtt{M}\ \overrightarrow{\mathtt{N}}|\!)_\rho)$.

The size $|\mathtt{v}|$ of a value $\mathtt{v}$ is defined by $|\mathtt{c}| = 1$ and $|\mathtt{M\ N}| = |\mathtt{M}| + |\mathtt{N}|$.

Lemma 6. *For any value* $\mathtt{v}$, *such that* $\emptyset; \Delta \vdash \mathtt{v} :: \mathtt{b}$, *with* $\mathtt{b} \in \boldsymbol{B}$, *we have* $(\!|\mathtt{v}|\!) = |\mathtt{v}|$.

Example 3. *Consider the following term* $\mathtt{M} :: \mathtt{Nat} \longrightarrow \mathtt{Nat}$ *computing the double of a unary number given as input:*

$$
\begin{aligned}
\mathtt{letRec\ f} = \lambda \mathtt{x.case\ x\ of\ } &+\mathtt{1(y)} \to +\mathtt{1}(+\mathtt{1(f\ y)}) \\
&\mid 0 \to 0
\end{aligned}
$$

We can see in Figure **??** *how the interpretation rules of Figure* **??** *are applied on such a term. At the end we search for the minimal strictly monotonic function* F *greater than* $\Lambda X.(5 \oplus (F\ (X-1))) \sqcup 4$, *for* $X > 1$. *That is* $\Lambda X.5X \oplus 4$. *Notice that such an interpretation is not tight as one should have expected the interpretation of such a program to be* $\Lambda X.2X \oplus k$ *for some constant* k. *This interpretation underlies that each iteration step, distinct from the base case, counts for 5 (1 for the recursive call, 1 for the application, 1 for pattern matching and 2 for the extra-constructors added) and the base case counts for 4 (letrec call, application, pattern matching and one constructor). Consequently, we have a bound on both size of terms and reduction length though this upper bound is not that accurate. This is not that surprising as this technique suffers from the same issues as methods based on first-order interpretations.*

3.5 Relaxing interpretations

For a given program it is somewhat difficult to find an interpretation that can be expressed in an easy way. This difficulty lies in the homogeneous definition of the considered interpretations using a max (for the case construct) and a min (for the letrec construct). Indeed, it is sometimes difficult to eliminate the constraint (parameters) of a max generated by the interpretation of a case. Moreover, it is a hard task to find the fixpoint of the interpretation of a letrec. All this can be relaxed as follows:

- finding an upper-bound of the max by eliminating constraint in the case construct interpretation,
- taking a particular function satisfying the inequality in the letrec construct interpretation.

In both cases, we will no longer compute an exact interpretation of the term but rather an upper bound of the interpretation.

Lemma 7. *Given a set of functions S and a function $F \in S$, the following inequality always holds $F \geq \sqcap\{G | G \in S\}$.*

This relaxation is highly desirable in order to find "lighter" upper-bounds on the interpretation of a term. Moreover, it is a reasonable approximation as we are interested in worst case complexity.

Obviously, it is possible by relaxing too much to attain the interpretation $\top_{(\!|\mathtt{T}|\!)}$, which gives nothing interesting.

Example 4. *Consider the term* `M` *of Example* **??***:*

$$\begin{aligned}\mathtt{letRec\ f} = \lambda g.\lambda \mathtt{x.case\ x\ of\ c(y,z)} &\rightarrow \mathtt{c}(g\ \mathtt{y})(\mathtt{f}\ g\ \mathtt{z})\\ \mid \mathtt{nil} &\rightarrow \mathtt{nil}\end{aligned}$$

We can see in the first four lines of Figure **??** *how the interpretation rules of Figure* **??** *are applied on such a term.*

In the penultimate line, we obtain an upper-bound on the interpretation by approximating the case interpretation, substituting $(\!|x|\!)_\rho - 1$ to both $(\!|y|\!)_\rho$ and $(\!|z|\!)_\rho$. This is the first step of relaxation where we find an upper bound for the max. The inequality:

$$a = \sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho + (\!|\mathtt{z}|\!)_\rho} F((\!|\mathtt{y}|\!)_\rho, (\!|\mathtt{z}|\!)_\rho) \leq \sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho, (\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{z}|\!)_\rho} F((\!|\mathtt{y}|\!)_\rho, (\!|\mathtt{z}|\!)_\rho) = b$$

holds for any monotonic function F. Consequently, $\sqcup\{F \mid F \geq b\} \leq \sqcup\{F \mid F \geq a\}$.

In the last line, we obtain an upper-bound on the interpretation by approximating the letrec interpretation, just checking that the function $\Lambda(\!|g|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(5 \oplus ((\!|g|\!)_\rho\ (\!|\mathtt{x}|\!)_\rho)) \times (\!|\mathtt{x}|\!)_\rho$, where $\times$ is the usual multiplication symbol over natural numbers, satisfies the inequality $F \geq 5 \oplus (\Lambda(\!|g|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(((\!|g|\!)_\rho\ ((\!|\mathtt{x}|\!)_\rho - 1)) + (F\ (\!|g|\!)_\rho\ ((\!|\mathtt{x}|\!)_\rho - 1))))$.

3.6 Higher-Order Polynomial Interpretations

At the present time, the interpretation of a term of type `T` can be any total functional over $(\!|\mathtt{T}|\!)$. In the next section, we will concentrate our efforts to study polynomial time at higher order. Consequently, we need to restrict the shape of the admissible interpretations. For that purpose, we introduce higher-order polynomials which are the higher-order counterpart to polynomials in this theory of complexity.

$$
\begin{aligned}
&(\!|\mathtt{M}|\!)_\rho \\
&= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 1 \oplus (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\lambda \mathtt{g}.\lambda \mathtt{x}.\mathtt{case\ x\ of\ c(y,z)} \to \mathtt{c(g\ y)(f\ g\ z)} | \ \mathtt{nil} \to \mathtt{nil}|\!)_\rho)F\} \\
&= \sqcap\{F \mid F \geq 3 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.\Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(\!|\mathtt{case\ x\ of\ c(y,z)} \to \mathtt{c(g\ y)(f\ g\ z)} | \ \mathtt{nil} \to \mathtt{nil}|\!)_\rho)F)\} \\
&= \sqcap\{F \mid F \geq 4 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.\Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho. \sqcup ((\!|\mathtt{nil}|\!)_\rho, \sqcup_{(\!|\mathtt{x}|\!)_\rho \geq (\!|\mathtt{c(y,z)}|\!)_\rho}((\!|\mathtt{c(g\ y)(f\ g\ z)}|\!)_\rho)))F)\} \\
&= \sqcap\{F \mid F \geq 4 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.\Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho. \sqcup (1, \sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho + (\!|\mathtt{z}|\!)_\rho}(1 \oplus ((\!|\mathtt{g}|\!)_\rho\ (\!|\mathtt{y}|\!)_\rho) \\
&\quad + ((\!|\mathtt{f}|\!)_\rho\ (\!|\mathtt{g}|\!)_\rho\ (\!|\mathtt{z}|\!)_\rho))))F)\} \\
&= \sqcap\{F \mid F \geq 5 \oplus (\Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho. \sqcup_{(\!|\mathtt{x}|\!)_\rho \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho + (\!|\mathtt{z}|\!)_\rho} (((\!|\mathtt{g}|\!)_\rho\ (\!|\mathtt{y}|\!)_\rho) + (F\ (\!|\mathtt{g}|\!)_\rho\ (\!|\mathtt{z}|\!)_\rho)))\} \\
&\leq \sqcap\{F \mid F \geq 5 \oplus (\Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(((\!|\mathtt{g}|\!)_\rho\ ((\!|\mathtt{x}|\!)_\rho - 1)) + (F\ (\!|\mathtt{g}|\!)_\rho\ ((\!|\mathtt{x}|\!)_\rho - 1))))\} \\
&\leq \Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(5 \oplus ((\!|\mathtt{g}|\!)_\rho\ (\!|\mathtt{x}|\!)_\rho)) \times (\!|\mathtt{x}|\!)_\rho
\end{aligned}
$$

Figure 4: Example of interpretation

Definition 3. *Let P_i denote a polynomial of order i and let X_i denote an order i variable. Higher-order (order 1 and order i+1) polynomials can be defined by the following grammar:*

$$
\begin{aligned}
P_1 &::= c | X_0 | P_1 + P_1 | P_1 \times P_1 \\
P_{i+1} &::= P_i | P_{i+1} + P_{i+1} | P_{i+1} \times P_{i+1} | X_i(P_{i+1})
\end{aligned}
$$

where c represents constants in $\mathbb{N}$.

Clearly, the set of order i polynomials is strictly included in the set of order $i+1$ polynomials by the above definition. Moreover, by definition, a higher-order polynomial P_{i+1} has variables of order at most i. If $\overrightarrow{X}$ is the sequence of such variables ordered by decreasing order, we will treat the polynomial P_{i+1} as total functions $\Lambda\overrightarrow{X}.P_{i+1}(\overrightarrow{X})$.

We are now ready to define the class of functions computed by terms admitting an interpretation that is (higher-order) polynomially bounded:

Definition 4. *Let FP_i, $i > 0$, be the class of polynomial functionals at order i that consist in functionals computed by closed terms $\mathtt{M}$ over the basic type $\mathcal{C}_{\mathtt{Nat}}$ and such that:*

- $\mathtt{ord}(\mathtt{M}) = i$
- $(\!|\mathtt{M}|\!)_\rho$ *is bounded by an order i polynomial (i.e. $\exists P_i,\ (\!|\mathtt{M}|\!)_\rho \leq P_i$).*

4 Safe Feasible Functionals

The class of tractable type 2 functionals has been introduced by Constable [**?**] and Mehlhorn [**?**]. It was later named BFF for the class of Basic Feasible Functionals and characterized in terms of function algebra [**?**, **?**]. We choose to define the class through a characterization by Bounded Typed Loop Programs from [**?**] which extends the original BFF to any order.

4.1 Basic Feasible Functionals and Bounded Typed Loop Programs

Definition 5 (BTLP)**.** *A Bounded Typed Loop Program (BTLP) is a non-recursive and well-formed procedure defined by the grammar of Figure* **??**.

$$
\begin{array}{rl}
\text{(Procedures)} \ni P & ::= \textbf{Procedure } v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})\ P^*\ V\ I^*\ \textbf{Return } v_r^{\mathtt{D}} \\
\text{(Declarations)} \ni V & ::= \textbf{Var } v_1^{\mathtt{D}},\ldots,v_n^{\mathtt{D}}; \\
\text{(Instructions)} \ni I & ::= v^{\mathtt{D}} := E; \mid \textbf{Loop } v_0^{\mathtt{D}} \textbf{ with } v_1^{\mathtt{D}} \textbf{ do } \{I^*\} \\
\text{(Expressions)} \ni E & ::= 1 \mid v^{\mathtt{D}} \mid v_0^{\mathtt{D}} + v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}} - v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}} \# v_1^{\mathtt{D}} \mid v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(A_1^{\tau_1},\ldots,A_n^{\tau_n}) \\
\text{(Arguments)} \ni A & ::= v \mid \lambda v_1,\ldots,v_n.v(v'_1\ldots,v'_m) \quad \text{with } v\notin\{v_1,\ldots,v_n\}
\end{array}
$$

Figure 5: BTLP grammar

The well-formedness assumption is given by the following constraints: Each procedure is supposed to be well-typed with respect to simple types over $\mathtt{D}$, the set of natural numbers of dyadic representation over $\{0,1\}$ ($0\equiv\epsilon$, $1\equiv 0$, $2\equiv 1$, $3\equiv 00$, ...). When needed, types are explicitly mentioned in variables' superscript. Each variable of a BTLP procedure is bound by either the procedure declaration parameter list, a local variable declaration or a lambda abstraction. In a loop statement, the guard variables v_0 and v_1 cannot be assigned to within I^*. In what follows v_1 will be called the *loop bound*.

The operational semantics of BTLP procedures is standard: parameters are passed by call-by-value. $+$, $-$ and $\#$ denote addition, proper subtraction and smash function (i.e. $x\#y = 2^{|x|\times|y|}$, the size $|x|$ of the number x being the size of its dyadic representation), respectively. Each loop statement is evaluated by iterating $|v_0|$-many times the loop body instruction under the following restriction: if an assignment $v := E$ is to be executed within the loop body, we check if the value obtained by evaluating E is of size smaller than the size of the loop bound $|v_1|$. If not then the result of evaluating this assignment is to assign 0 to v.

We are now ready to provide a definition of Basic Feasible Functionals at any order:

Definition 6 ($\textsc{bff}_i$)**.** *For any $i\geq 1$, $\textsc{bff}_i$ is the class of order i functionals computable by a BTLP procedure*[1]*.*

It is straightforward that $\textsc{bff}_1 = \textsc{FPtime}$ (the class of first-order polynomial time computable functions) and $\textsc{bff}_2$=BFF.

4.2 Safe Feasible Functionals

Now we restrict the domain of $\textsc{bff}_i$ classes to inputs in $\textsc{bff}_k$ for $k<i$, the obtained classes are named SFF for Safe Feasible Functionals.

Definition 7 ($\textsc{sff}_i$)**.** $\textsc{sff}_1$ *is defined to be the class of order* 1 *functionals computable by BTLP a procedure and, for any $i\geq 1$, $\textsc{sff}_{i+1}$ is the class of order $i+1$ functionals computable by BTLP a procedure on the input domain $\textsc{sff}_i$. In other words,*

$$
\begin{aligned}
\textsc{sff}_1 &= \textsc{bff}_1, \\
\forall i\geq 1,\ \textsc{sff}_{i+1} &= \textsc{bff}_{i+1\restriction\textsc{sff}_i}
\end{aligned}
$$

This is not a huge restriction since we want an arbitrary term of a given complexity class at order i to compute over terms that are already in classes of the same family at order k, for

[1] As demonstrated in [**?**], all types in the procedure can be restricted to be of order at most i without any distinction.

$k < i$. Consequently, programs can be built in a constructive way component by component. Another strong argument in favor of this domain restriction is that the partial evaluation of a functional at order i will, at the end, provide a function in $\mathbb{N} \longrightarrow \mathbb{N}$ that is shown to be in BFF_1 (=FPTIME).

5 A characterization of Safe Feasible Functionals of any order

Theorem 1. *For any order $i \geq 1$, the class of functions in FP_i over FP_k, $k < i$, is exactly the class of functionals in SFF_i. In other words, $\text{SFF}_i \equiv \text{FP}_{i\restriction(\cup_{k\leq i}\text{FP}_k)}$, for all $i \geq 1$.*

Proof. This Theorem is proved by induction on the order. For any order i, it can be split in two directions called *Soundness*(i) and *Completeness*(i):

- Soundness(i): Any term $\mathtt{M}$ whose interpretation is bounded by an order i polynomial, computes a function in SFF_i.
- Completeness(i): Any BTLP procedure P of order i can be encoded by a term $\mathtt{M}$ computing the same function and admitting a polynomial interpretation of order i.

We start to show that *Soundness*(1) holds. Consider that the term $\mathtt{M}$ has an interpretation bounded by a polynomial P_1. For any value v, we have, by Corollary **??**, that the computing time of $\mathtt{M}$ on input v bounded by $(\!|\mathtt{M}\ v|\!)_\rho$. Consequently, using Lemma **??**, we have that:

$$(\!|\mathtt{M}\ v|\!)_\rho = (\!|\mathtt{M}|\!)_\rho(\!|v|\!)_\rho = (\!|\mathtt{M}|\!)_\rho(|v|) \leq P_1(|v|).$$

Hence $\mathtt{M}$ belongs to $\text{FPTIME} = \text{SFF}_1$.

Completeness(1) holds as a direct consequence of Theorem 4 of [**?**] showing the completeness of polynomial interpretations of Kind(0) for confluent TRS with respect to the complexity class FPTIME. Kind(0) means that constructor symbol interpretations are enforced to be linear functions of the shape $\Sigma_{i=1}^{n} X_i + \alpha$, $\alpha \geq 1$. Just notice that interpretations in our framework are of Kind(0) and that each term rewrite rule admitting an interpretation can be encoded easily into an equivalent term of order 1 admitting an interpretation (up to some additive constant).

Now, we suppose that *Soundness*(k) and *Completeness*(k) hold for any $k \leq i$. Let $\mathtt{M}$ be an order $i+1$ term of interpretation $(\!|\mathtt{M}|\!)_\rho$ bounded by the order $i+1$ polynomial P_{i+1}, i.e. $(\!|\mathtt{M}|\!)_\rho \leq P_{i+1}$. We know that on input $\mathtt{N}$, $\mathtt{M}$ normalizes in $O((\!|\mathtt{M}\ \mathtt{N}|\!)_\rho)$, by Corollary **??**. Since $\mathtt{N}$ computes a functional $[\![\mathtt{N}]\!] \in \text{SFF}_i$, by induction (*Completeness*(i)) there is a polynomial P_i such that $(\!|\mathtt{N}|\!)_\rho \leq P_i$. Consequently, we obtain that the maximal number of reduction steps is bounded polynomially in the input size by:

$$(\!|\mathtt{M}\ \mathtt{N}|\!)_\rho = (\!|\mathtt{M}|\!)_\rho(\!|\mathtt{N}|\!)_\rho \leq P_{i+1} \circ P_i$$

that is, by a polynomial Q_{i+1} of order $i+1$ defined by $Q_{i+1} = P_{i+1} \circ P_i$. Consequently, *Soundness*($i+1$) holds. For *Completeness*($i+1$), we demonstrate using program transformations that any loop can be encoded by a first-order term whose runtime remains polynomial in the input size. For that purpose, we proceed in two steps:

1. first, we transform each BTLP procedure into a standard loop-based higher-order imperative procedure with explicit bounds, no nested loops and no procedure call. Instead of being checked dynamically the "with $v_1^{\mathtt{D}}$" bound is performed using a local `chkbd` operator

BTLP	Standard loop-based procedure

`Loop x with w {`
`  `I_1^*
`  Loop y with z {`
`  `I_2^*
`  }`
`  `I_3^*
`}`
	`total := w×y; dx := 1;`
	`gt := total; gy := y;`
	`lb := x#total;`
	`loop lb {`
	`  if dx {`$(I_1^*)_w$`}`
	`  if gy {(`$(I_2^*)_z$`)`$_w$ `gy := cut(gy);}`
	`  if dx {dx :=` ε`;` $(I_3^*)_w$`}`
	`  if gt {gt := cut(gt);}`
	`  else {gt := total; gy := y; dx := 1;}`
	`}`

Figure 6: Rewriting nested loops. The variables dx, total, gt, gy and lb are fresh local variables.

BTLP	Standard loop-based procedure

`Loop` x_1 `with` w_1 `{`
`  `I_1^*
`}`
I_2^*
`Loop` x_3 `with` w_3 `{`
`  `I_3^*
`}`
	`gx := x1; dy := 1; lb :=` $x_1 \times x_3$`;`
	`loop lb {`
	`  if gx {gx := cut(gx);` $(I_1^*)_{w_1}$`}`
	`  else {`
	`  if dy {dy :=` ε`;` I_2^*`}`
	`  else {`$(I_3^*)_{w_3}$`}`
	`  }`
	`}`

Figure 7: Rewriting sequential loops. Variables gx, dy and lb are fresh variables.

in SFF_1 provided that the input E is computable in polynomial time and whose semantics is defined by:

$$\texttt{chkbd}(E, X) = \begin{cases} [\![E]\!] & \text{if } |[\![E]\!]| \leq |x|, x \in X \\ 0 & \text{otherwise} \end{cases}$$

where $[\![E]\!]$ is the dyadic number obtained after the evaluation of expression E and X is a finite set of variables. For example, **Loop** x **with** w **do** $\{y := y + z;\}$ is transformed into **loop** $x\{y :=$ `chkbd`$(y + z, \{w\});\}$. Figures **??** and **??** present some translations to get rid of, respectively, nested loops and sequential loops. They use the same operators as BTLP (in particular `#`), but also standard multiplication $\times$. They also use a new `cut` operator that is defined as removing the first character of its argument. Last, the notation $(I)_v$ which propagates to all assignments in I translating them using `chkbd` such as $x := E$ (resp. $x :=$ `chkbd`(E, X)) being translated into $x :=$ `chkbd`$(E, \{v\})$ (resp. $x :=$ `chkbd`$(E, \{v\} \cup X)$). Procedure calls are eliminated by unfolding procedure bodies after a careful alpha-conversion. Nested loop elimination terminates as each application consumes one loop while unfolding terminates as there are no recursive calls by definition of BTLP. Notice that this program transformation preserves both the semantics and the asymptotic complexity of the initial BTLP procedure.

2. second, we compile each standard loop-based higher-order imperative procedure into a

functional term. For that purpose, we suppose that the target term language includes constructors for dyadic numbers (ϵ, 0 and 1), a constructor for tuples $\langle \ldots \rangle$, all the imperative operators as basic prefix operators (+, -, #, ...), min operators $\sqcap^{\mathtt{p}}_n$ computing the min of the sizes of n dyadic numbers and a `chkbd` operator of arity 2 such that $[\![\mathtt{chkbd}(\mathtt{M},\mathtt{N})]\!] = [\![\mathtt{M}]\!]$ if $|[\![\mathtt{M}]\!]| \leq |[\![\mathtt{N}]\!]|$ (and ϵ otherwise). All these operators are extended to be total functions in the term language: they return ϵ on input terms for which they are undefined. Moreover, we also require that each procedure given as input is alpha renamed so that all parameters and local variables of a given procedure have the same name and are indexed by natural numbers. The compiling process is described in Figure **??**, defining $\mathtt{comp}, \mathtt{comi}^n, \mathtt{come}$ that respectively compile procedures, instructions and expressions. The `comi` compiling process is indexed by the number of variables in the program n. For notational convenience, let $\lambda\langle v_1, \ldots, v_n\rangle.$ be a shorthand notation for $\lambda s.\mathtt{case}\ s\ \mathtt{of}\ \langle v_1, \ldots, v_n\rangle \rightarrow$ and let π^n_r be a shorthand notation for the r-th projection $\lambda\langle v_1, \ldots, v_n\rangle.v_r$. The compilation procedure works as follows:

- Any loop-based procedure of the shape

$$v(v_1, \ldots, v_m)\ \mathbf{Var}\ v_{m+1}, \ldots, v_n;\ I^*\ \ \mathbf{Return}\ v_r$$

  will be transformed into a term of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_r$, provided that τ_i is the type of v_i and that $\tau_1 \times \ldots \times \tau_n$ is the type for n-ary tuples of the shape $\langle v_1, \ldots, v_n\rangle$.
- Each instruction within a procedure of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_r$ will have type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_1 \times \ldots \times \tau_n$. Consequently, two sequential instructions $I_1\ I_2$ within a procedure of n variables will be compiled into a term application of the shape $\mathtt{comi}^n(I_2)\ \mathtt{comi}^n(I_1)$ and instruction type is preserved by composition.
- For loop compilation, we make use of a letrec of type $\mathtt{D} \rightarrow \tau_1 \times \ldots \times \tau_n \rightarrow \tau_1 \times \ldots \times \tau_n$. The first argument is a counter and is fed with a copy of the loop counter v_i so that the obtained term has the expected type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_1 \times \ldots \times \tau_n$.

Notice that this compilation preserves both the semantics and its asymptotic complexity of the initial BTLP procedure.

Finally, we have transformed a BTLP procedure into a term preserving the semantics and the asymptotic complexity (reduction length). Moreover, each higher-order expression in a tuple can be encoded by a first-order term using defunctionalization and we let the reader check that the operators in $\{+, -, \#, \times, \mathtt{cut}, \sqcap^{\mathtt{p}}_n, \mathtt{chkbd}\}$ admit the following polynomial sup-interpretations $(\!|+|\!)_\rho = \Lambda X.\Lambda Y.X + Y$, $(\!|-|\!)_\rho = \Lambda X.\Lambda Y.X$, $(\!|\#|\!)_\rho = \Lambda X.\Lambda Y.X \times Y$, $(\!|\times|\!)_\rho = \Lambda X.\Lambda Y.X + Y$, $(\!|\mathtt{cut}|\!)_\rho = \Lambda X.\sqcup(X-1, 0)$, $(\!|\sqcap^{\mathtt{p}}_n|\!)_\rho = \Lambda X_1.\ldots.\Lambda X_n.\sqcap^{\overline{\mathbb{N}}}(X_1, \ldots, X_n)$, $(\!|\mathtt{chkbd}|\!)_\rho = \Lambda X.\Lambda Y.Y$. By *Completeness*(1), there exists a term computing the same function and admitting a polynomial interpretation. □

Consider the following simple BTLP procedure as an illustrative example for the program transformation in the proof of Theorem **??**:

```
Procedure mult(x,y)
  Var z,b; z := ϵ; b := x#y; Loop x with b do{z := z+y;}
  Return z
```

The transformation process is easy as there is no procedure call and no nested loop and we obtain the following loop program:

```
Procedure mult(x,y)
    var z,b; z := ϵ; b := x#y; loop x {z := chkbd(z+y,{b});}
    Return z
```

$$\texttt{comp}(\textbf{Procedure } v(v_1,\dots,v_m) \textbf{ Var } v_{m+1},\dots,v_n;\ I^* \textbf{ Return } v_r) = \lambda s.\pi_r^n(\texttt{comi}^n(I^*)\ s)$$
$$\texttt{comi}^n(I_1\ \dots\ I_k) = \texttt{comi}^n(I_k)\ \dots\ \texttt{comi}^n(I_1)$$
$$\texttt{comi}^n(v_i := E;) = \lambda\langle v_1,\dots,v_n\rangle.\langle v_1,\dots,v_{i-1},\texttt{come}(E),v_{i+1},\dots,v_n\rangle$$
$$\texttt{comi}^n(\textbf{loop } v_i\ \{I^*\}) = \lambda\langle v_1,\dots,v_n\rangle.\left(\left(\begin{array}{rcl}\texttt{letRec } f = \lambda\tilde{t}.\lambda s.\texttt{case } \tilde{t} \texttt{ of } \epsilon & \to & s\\ & & j(t) \to f\ t\ (\texttt{comi}^n(I^*)\ s)\end{array}\right) v_i\right)$$
$$\texttt{comi}^n(\textbf{if } v_i^{\mathtt{D}}\ \{\ I_1^*\ \}\ \textbf{else}\ \{\ I_2^*\}) = \lambda\langle v_1,\dots,v_n\rangle.\texttt{case } v_i \texttt{ of } \epsilon \to \texttt{comi}^n(I_2^*) \mid j(t) \to \texttt{comi}^n(I_1^*)$$
with $j \in \{0,1\}$
$$\texttt{come}(c) = c, \quad c \in \{1, v^{\mathtt{D}}\}$$
$$\texttt{come}(v_0^{\mathtt{D}}\ \texttt{op}\ v_1^{\mathtt{D}}) = \texttt{op}\ v_0\ v_1, \quad \texttt{op} \in \{+,-,\#,\times\}$$
$$\texttt{come}(\texttt{cut}(v^{\mathtt{D}})) = \texttt{cut}\ v$$
$$\texttt{come}(\texttt{chkbd}(E,\{v_{j_1},\dots,vj_r\})) = \texttt{chkbd come}(E)\ (\sqcap_r^{\mathtt{D}}\ v_{j_1}\ \dots\ v_{j_r})$$
$$\texttt{come}(v(A_1,\dots,A_n)) = v\ \texttt{come}(A_1)\ \dots\ \texttt{come}(A_n)$$
$$\texttt{come}(\lambda v_1,\dots,v_n.v(v_1'\dots,v_m')) = \lambda v_1.\dots\lambda v_n.(\dots(v\ v_1')\ \dots\ v_m')\dots)$$

Figure 8: Compiler from loop programs to terms

As already stated, the semantics and asymptotic complexity remain preserved and, in a next step, this loop program is compiled modulo α-equivalence into:

$$\lambda s.\pi_3^4\left(\begin{array}{l}\lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\left(\texttt{letRec } f = \begin{array}{rcl}\lambda\tilde{t}.\lambda\tilde{s}.\texttt{case } \tilde{t} \texttt{ of } \varepsilon & \to & s\\ & & j(t) \to f\ t\ (\mathtt{M}\ \tilde{s}))\end{array}\right)\mathtt{x}\\ (\lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\langle \mathtt{x},\mathtt{y},\mathtt{z},\#\ \mathtt{x}\ \mathtt{y}\rangle(\lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\langle \mathtt{x},\mathtt{y},\epsilon,b\rangle\ s))\end{array}\right)$$

where $\mathtt{M} = \lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\langle \mathtt{x},\mathtt{y},\mathtt{chkbd}\ (+\ \mathtt{z}\ \mathtt{y})\ (\sqcap_1^{\mathtt{D}}\ \mathtt{b}),\mathtt{b}\rangle$. The semantics and asymptotic complexity are still preserved as the recursive call iterates at most $|x|$ times.

6 Results and perspectives

In this paper, we have introduced a theory for higher-order interpretations that can be used to deal with higher-order complexity classes. This is a novel approach but there are still some important issues to discuss.

- Expressivity: the expressivity of interpretations is as usual their main drawback: As for first-order interpretations, a lot of interesting terms computing polynomial time functions will not have any polynomial interpretation, *i.e.* its interpretation will sometimes be $\top$, although the function will be computed by another algorithm (term) admitting a finite interpretation. It should be possible to relax the tool and to extend it to more complex data structures such as streams and infinite trees. At least, the presented paper has shown that the tool expressivity can be extended to higher order. Moreover, we manage to give a uniform framework for complexity analysis of exponential programs behaving polynomially. As an illustrative example, consider the term M by:

$$\texttt{letRec fold} = \lambda\mathtt{g}.\lambda\mathtt{x}.\texttt{case x of c}(\mathtt{y},\mathtt{z}) \to \mathtt{g}\ (\texttt{fold g z}) \mid \texttt{nil} \to \texttt{nil}$$

The interpretation of $\mathtt{M}$ has to satisfy the following inequality:

$$\sqcap\{F \mid F \geq \Lambda G.\Lambda X.4 \oplus (1 \sqcup (\sqcup_{X \geq 1}(G(F\ G\ (X-1)))))\}$$

Clearly, this inequality does not admit any polynomial interpretation as it is at least exponential in X. Now consider the term $\mathtt{M}\ (\lambda \mathtt{x}.1 + (1 + (\mathtt{x}))$. The term $\lambda \mathtt{x}.1 + (1 + (\mathtt{x}))$ can be given the interpretation $\Lambda X.X \oplus 3$. We have to find a function F such that $F\ (\Lambda X.X \oplus 3) \geq \Lambda Y.4 \oplus (1 \sqcup (\sqcup_{Y \geq 1}(F\ (\Lambda X.X \oplus 3)\ (Y-1)) \oplus 3))$. This inequality is satisfied by the function F such that $F\ (\Lambda X.X \oplus 3)\ Y = (7 \times Y) \oplus 4$ and consequently $\mathtt{M}$ has an interpretation. This highlights the fact that a term may have an interpretation even if some of its subterms might not have any. As expected, any term admitting an interpretation of the shape $\Lambda X.X \oplus \beta$, for some constant β, will have an interpretation when applied as first operand of this fold function.

- Synthesis: it has been well known for a long time that the synthesis problem that consists in finding the sup-interpretation of a given term is undecidable in general for first-order terms using interpretations over natural numbers (see [**?**] for a survey). As a consequence this problem is also undecidable for higher order. However it is also well-known that it will become decidable if programs are defunctionalized (up to a first-order term) using polynomial functions over real numbers.
- Space: space issues were not discussed in this paper as there is no theory for higher-order polynomial space. In analogy with the usual first-order theory, a suitable definition for higher-order space complexity classes could be to consider terminating terms admitting a nonstrict polynomial interpretation. By nonstrict, we mean, for example, that the last rule of Figure **??** can be replaced by:

$$(\!|\mathtt{letRec\ f} = \mathtt{M}|\!)_\rho = 1 \oplus \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq (\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{M}|\!)_\rho)F\}$$

Thus termination is lost as the term $\mathtt{letRec\ f} = \mathtt{f}$ could be interpreted by $1 \oplus \Lambda F.F$. This is the reason why we need to restrict the class to terminating terms. However a result equivalent to Lemma **??** holds: we still keep a non-strict upper bound on the interpretation of any derived term.

Acknowledgments. The authors would like to thank the anonymous reviewers for their comments and suggestions to improve this paper.

References

[1] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Inf. Comput.*, 248:56–81, 2016.

[2] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *J. Funct. Program.*, 11(1):33–53, 2001.

[3] Robert L. Constable. Type two computational complexity. In *Proc. 5th annual ACM Symposium on Theory of Computing*, pages 108–121. ACM, 1973.

[4] Stephen A. Cook and Bruce M. Kapron. Characterizations of the basic feasible functionals of finite type. In *Foundations of Computer Science*, pages 154–159. IEEE Computer Society, 1989.

[5] Hugo Férée. *Complexité d'ordre supérieur et analyse récursive. (Higher order complexity and computable analysis)*. PhD thesis, University of Lorraine, Nancy, France, 2014.

[6] Robert J. Irwin, Bruce M. Kapron, and James S. Royer. On characterizations of the basic feasible functionals (part II). Technical report, Syracuse University, 2002.

[7] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. On characterizations of the basic feasible functionals (part I). *Journal of Functional Programming*, 11(1):117–153, 2001.

[8] Akitoshi Kawamura and Stephen A. Cook. Complexity theory for operators in analysis. In Leonard J. Schulman, editor, *Proc. 42nd ACM Symposium on Theory of Computing*, pages 495–502. ACM, 2010.

[9] William A. Kirk and Brailey Sims. *Handbook of metric fixed point theory*. Springer, 2001.

[10] Ker-I Ko. *Complexity Theory of Real Functions*. Birkhäuser, 1991.

[11] Dallas S. Lankford. On proving term rewriting systems are noetherian. Technical report, 1979.

[12] Zohar Manna and Steven Ness. On the termination of Markov algorithms. In *Third Hawaii International Conference on System Science*, pages 789–792, 1970.

[13] Kurt Mehlhorn. Polynomial and abstract subrecursive classes. In *Proc. 6th annual ACM Symposium on Theory of Computing*, pages 96–109. ACM, 1974.

[14] Romain Péchoux. Synthesis of sup-interpretations: a survey. *Theor. Comput. Sci.*, 467:30–52, 2013.

[15] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[16] Jaco Van de Pol. Termination proofs for higher-order rewrite systems. In *International Workshop on Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *LNCS*, pages 305–325. Springer, 1993.

[17] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.