Theory of Higher Order Interpretations and Application to Basic Feasible Functions

Emmanuel Hainry, Romain Péchoux

THEORY OF HIGHER ORDER INTERPRETATIONS AND APPLICATION TO BASIC FEASIBLE FUNCTIONS

EMMANUEL HAINRY AND ROMAIN PÉCHOUX

Inria project Mocqua, CNRS, and Université de Lorraine, LORIA
e-mail address: emmanuel.hainry@loria.fr

Inria project Mocqua, CNRS, and Université de Lorraine, LORIA
e-mail address: romain.pechoux@loria.fr

ABSTRACT. Interpretation methods and their restrictions to polynomials have been deeply used to control the termination and complexity of first-order term rewrite systems. This paper extends interpretation methods to a pure higher order functional language. We develop a theory of higher order functions that is well-suited for the complexity analysis of this programming language. The interpretation domain is a complete lattice and, consequently, we express program interpretation in terms of a least fixpoint. As an application, by bounding interpretations by higher order polynomials, we characterize Basic Feasible Functions at any order.

1. INTRODUCTION

1.1. **Higher order interpretations.** This paper introduces a theory of higher order interpretations. These interpretations are an extension of usual (polynomial) interpretation methods introduced in [MN70, Lan79] and used to show the termination of (first order) term rewrite systems [CMPU05, CL92] or to study their complexity [BMM11].

This theory is a novel and uniform extension to higher order functional programs: the definition works at any order on a simple programming language, where interpretations can be elegantly expressed in terms of a least fixpoint, and no extra constraints are required.

The language has only one semantics restriction: its reduction strategy is enforced to be leftmost outermost as interpretations are non decreasing functions. Similarly to first order interpretations, higher order interpretations ensure that each reduction step corresponds to a strict decrease. Consequently, some of the system properties could be lost if a reduction occurs under a context.

This work has been partially supported by ANR Project ELICA ANR-14-CE25-0005.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

1.2. **Application to higher order polynomial time.** As Church-Turing's thesis does not hold at higher order, distinct and mostly pairwise uncomparable complexity classes are candidates as a natural equivalent of the notion of polynomial time computation for higher order.

The class of polynomially computable real functions by Ko [Ko91] and the class of Basic Feasible Functional at order i (BFF_i) by Irwin, Kapron and Royer [IKR02] belong to the most popular definitions for such classes. In [Ko91] polynomially computable real functions are defined in terms of first order functions over real numbers. They consist in order 2 functions over natural numbers and an extension at any order is proposed by Kawamura and Cook in [KC10]. The main distinctions between these two models are the following:

- the paper [Ko91] deals with representation of real numbers as input while the paper [IKR02] deals with general functions as input,
- the paper [Ko91] deals with the number of steps needed to produce an output at a given precision while the paper [IKR02] deals with the number of reduction steps needed to evaluate the program.

Moreover, it was shown in [IKR02] and [Fér14] that the classes BFF_i cannot capture some functions that could be naturally considered to be polynomial time computable because they do not take into account the size of their higher order arguments. However they have been demonstrated to be robust, they characterize exactly the well-known class FPTIME of polynomial time computable functions as well as the well-known class of Basic Feasible Functions BFF, that corresponds to order 2 polynomial time computations, and have already been characterized in various ways, *e.g.* [CK89].

The current paper provides a characterization of the BFF_i classes as they deal with discrete data as input and they are consequently more suited to be studied with respect to usual functional languages. This result was expected to hold as it is known for a long time that (first order) polynomial interpretations characterize FPTIME and as it is shown in [FHHP15] that (first order) polynomial interpretations on stream programs characterize BFF.

1.3. **Related works.** The present paper is an extended version of the results in [HP17]: more proofs and examples have been provided. An erratum has been provided: the interpretation of the case construct has been slightly modified so that we can consider non decreasing functions (and not only strictly increasing functions).

There are two lines of work that are related to our approach. In [VdP93], Van De Pol introduced higher order interpretation for showing the termination of higher order term rewrite systems. In [BL12, BL16], Baillot and Dal Lago introduce higher order interpretations for complexity analysis of term rewrite systems. While the first work only deals with termination properties, the second work is restricted to a family of higher order term rewrite systems called simply typed term rewrite systems. Our work can be viewed as an extension of [BL16] to functional programs and polynomial complexity at any order.

1.4. **Outline.** In Section 2, the syntax and semantics of the functional language are introduced. The new notion of higher order interpretation and its properties are described in Section 3. Next, in Section 4, we briefly recall the BFF_i classes and their main characterizations, including a characterization based on the BTLP programming language of [IKR02]. Section 5 is devoted to the 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: compiling procedures to terms after applying some program transformations. In Section 6, we briefly discuss the open issues and future works related to higher order interpretations.

$$\frac{\mathtt{M} \Rightarrow^k \mathtt{N}}{\mathtt{case\ M\ of\ \ldots} \Rightarrow^k \mathtt{case\ N\ of\ \ldots}} \qquad \frac{\mathtt{M} \Rightarrow^k \mathtt{M'}}{\mathtt{M\,N} \Rightarrow^k \mathtt{M'\,N}} \qquad \frac{\mathtt{M} \rightarrow_\alpha \mathtt{N}}{\mathtt{M} \Rightarrow^1 \mathtt{N}} \qquad \frac{\mathtt{M} \Rightarrow^k \mathtt{M'} \quad \mathtt{M'} \Rightarrow^{k'} \mathtt{N}}{\mathtt{M} \Rightarrow^{k+k'} \mathtt{N}}$$

Figure 1: Evaluation strategy

2. Functional language

2.1. **Syntax.** The considered programming language consists in an unpure lambda calculus with constructors, primitive operators, a `case` construct for pattern matching and a `letRec` instruction for function definitions that can be recursive. It is as an extension of PCF [Mit96] to inductive data types and it enjoys the same properties (confluence and completeness with respect to partial recursive functions for example).

The set of terms $\mathcal{T}$ of the language is generated by the following grammar:

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

where $\mathtt{c}, \mathtt{c}_1, \cdots, \mathtt{c}_n$ 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. $\mathtt{x}, \mathtt{f}$ are variables in $\mathcal{X}$ and $\overrightarrow{\mathtt{x}_i}$ is a sequence of $ar(\mathtt{c}_i)$ variables.

The free variables $FV(\mathtt{M})$ of a term $\mathtt{M}$ are defined as usual. Bounded variables are assumed to have distinct names in order to avoid name clashes. A closed term is a term $\mathtt{M}$ with no free variables, $FV(\mathtt{M}) = \emptyset$.
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 $\mathtt{N}_1, \cdots, \mathtt{N}_n$. The result of applying the substitution $\{\mathtt{N}_1/\mathtt{x}_1, \cdots, \mathtt{N}_n/\mathtt{x}_n\}$ to a term $\mathtt{M}$ is noted $\mathtt{M}\{\mathtt{N}_1/\mathtt{x}_1, \cdots, \mathtt{N}_n/\mathtt{x}_n\}$ or $\mathtt{M}\{\vec{\mathtt{N}}/\vec{\mathtt{x}}\}$ when the substituting terms are clear from the context.

2.2. **Semantics.** Each primitive operator `op` has a corresponding semantics $[\![\mathtt{op}]\!]$ fixed by the language implementation. $[\![\mathtt{op}]\!]$ is a total function from $\mathcal{T}^{ar(\mathtt{op})}$ to $\mathcal{T}$.[1]

We define the following relations between two terms of the language:

- β-reduction: $\lambda\mathtt{x.M\ N} \rightarrow_\beta \mathtt{M}\{\mathtt{N}/\mathtt{x}\}$,
- pattern matching: $\mathtt{case\ c}_j(\overrightarrow{\mathtt{N}_j})\ \mathtt{of\ \ldots\ c}_j(\overrightarrow{\mathtt{x}_j}) \rightarrow \mathtt{M}_j \ldots \rightarrow_{\mathtt{case}} \mathtt{M}_j\{\overrightarrow{\mathtt{N}_j}/\overrightarrow{\mathtt{x}_j}\}$,
- operator evaluation: $\mathtt{op\ M}_1 \ldots \mathtt{M}_n \rightarrow_{\mathtt{op}} [\![\mathtt{op}]\!](\mathtt{M}_1, \ldots, \mathtt{M}_n)$,
- fixpoint evaluation: $\mathtt{letRec\ f = M} \rightarrow_{\mathtt{letRec}} \mathtt{M}\{\mathtt{letRec\ f = M/f}\}$,

Let $\rightarrow_\alpha$ be defined as $\cup_{r \in \{\beta, \mathtt{case}, \mathtt{letRec}, \mathtt{op}\}} \rightarrow_r$. Let $\Rightarrow^k$ be the leftmost outermost (normal-order) evaluation strategy defined with respect to $\rightarrow_\alpha$ in Figure 1. The index k accounts for the number of $\rightarrow_\alpha$ steps fired during a reduction. Let $\Rightarrow$ be a shorthand notation for $\Rightarrow^1$.

Let $|\mathtt{M} \Rightarrow^k \mathtt{N}|$ be the number of reductions distinct from $\rightarrow_{\mathtt{op}}$ in a given a derivation $\mathtt{M} \Rightarrow^k \mathtt{N}$. $|\mathtt{M} \Rightarrow^k \mathtt{N}| \leq k$ always holds. $[\![\mathtt{M}]\!]$ is a notation for the term computed by $\mathtt{M}$ (if it exists), *i.e.* $\exists k, \mathtt{M} \Rightarrow^k [\![\mathtt{M}]\!]$ and $\nexists \mathtt{N}, [\![\mathtt{M}]\!] \Rightarrow \mathtt{N}$.

A (first order) value $\mathtt{v}$ is defined inductively by either $\mathtt{v} = \mathtt{c}$, if $ar(\mathtt{c}) = 0$, or $\mathtt{v} = \mathtt{c}\ \vec{\mathtt{v}}$, for $ar(\mathtt{c}) > 0$ values $\vec{\mathtt{v}}$, otherwise.

[1] Operators are total functions over terms and are not only defined on *"values"*, *i.e.* terms of the shape $\lambda\mathtt{x.M}$, so that we never need to reduce the operands in rule $\rightarrow_{\mathtt{op}}$. This will allow us to consider non decreasing operator interpretations in Definition 3.5 instead of strictly increasing operator interpretation.

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

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

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

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

Figure 2: Type system

2.3. **Type system.** Let **B** be a set of basic inductive types $\mathtt{b}$ described by their constructor symbol sets $\mathcal{C}_{\mathtt{b}}$. The set of simple types is defined by:

$$\mathtt{T} ::= \mathtt{b} \mid \mathtt{T} \longrightarrow \mathtt{T}, \quad \text{with } \mathtt{b} \in \mathbf{B}.$$

As usual $\longrightarrow$ associates to the right.

Example 2.1. The type of unary numbers $\mathtt{Nat}$ can be described by $\mathcal{C}_{\mathtt{Nat}} = \{0, +1\}$, 0 being a constructor symbol of 0-arity and $+1$ being a constructor symbol of 1-arity.

For any type T, $[T]$ is the base type for lists of elements of type $\mathtt{T}$ and has constructor symbol set $\mathcal{C}_{[\mathtt{T}]} = \{\mathtt{nil}, \mathtt{c}\}$, $\mathtt{nil}$ being a constructor symbol of 0-arity and $\mathtt{c}$ being a constructor symbol of 2-arity.

The type system is described in Figure 2 and proves judgments of the shape $\Gamma;\Delta \vdash \mathtt{M} :: \mathtt{T}$ meaning that the term $\mathtt{M}$ has type $\mathtt{T}$ under the variable and constructor symbol contexts Γ and Δ respectively ; a variable (a constructor, respectively) context being a partial function that assigns types to variables (constructors and operators, respectively).

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 $\mathtt{b}_1 \longrightarrow \ldots \longrightarrow \mathtt{b}_n \longrightarrow \mathtt{b}$. A well-typed term will consist in a term $\mathtt{M}$ such that $\emptyset;\Delta \vdash \mathtt{M} :: \mathtt{T}$. Consequently, it is mandatory for a term to be closed in order to be well-typed.

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.

Definition 2.2 (Order)**.** The order of a type $\mathtt{T}$, noted $\mathtt{ord}(\mathtt{T})$, is defined inductively by:

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

Given a term $\mathtt{M}$ of type $\mathtt{T}$, *i.e.* $\emptyset;\Delta \vdash \mathtt{M} :: \mathtt{T}$, the order of $\mathtt{M}$ with respect to $\mathtt{T}$ is $\mathtt{ord}(\mathtt{T})$.

Example 2.3. Consider the following term $\mathtt{M}$ that maps a function to a list given as inputs:

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

Let [Nat] is the base type for lists of natural numbers of constructor symbol set $\mathcal{C}_{[\mathtt{Nat}]} = \{\mathtt{nil}, \mathtt{c}\}$. The term M can be typed by $\emptyset; \Delta \vdash \mathtt{M} :: (\mathtt{Nat} \longrightarrow \mathtt{Nat}) \longrightarrow [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]$, as illustrated by the following typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\dots \quad \Gamma'; \Delta \vdash (\mathtt{g}\ \mathtt{y}) :: \mathtt{Nat}}\,(\mathtt{App})}{\dots \quad \Gamma'; \Delta \vdash \mathtt{c}\ (\mathtt{g}\ \mathtt{y}) :: [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]}\,(\mathtt{App}) \qquad \dfrac{\dots}{\Gamma'; \Delta \vdash (\mathtt{f}\ \mathtt{g}\ \mathtt{z}) :: [\mathtt{Nat}]}\,(\mathtt{App})}{\dots \quad \Gamma, \mathtt{y} :: \mathtt{Nat}, \mathtt{z} :: [\mathtt{Nat}]; \Delta \vdash \mathtt{c}\ (\mathtt{g}\ \mathtt{y})\ (\mathtt{f}\ \mathtt{g}\ \mathtt{z}) :: [\mathtt{Nat}]}\,(\mathtt{App})}{\Gamma, \mathtt{x} :: [\mathtt{Nat}]; \Delta \vdash \mathtt{case\ x\ of\ \ c(y,z) \to c\ (g\ y)\ (f\ g\ z), nil \to nil} :: [\mathtt{Nat}]}\,(\mathtt{Case})}{\Gamma; \Delta \vdash \lambda \mathtt{x.case\ x\ of\ \ c(y,z) \to c\ (g\ y)\ (f\ g\ z), nil \to nil\ } :: [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]}\,(\mathtt{Abs})}{\mathtt{f} :: \mathtt{T}; \Delta \vdash \lambda \mathtt{g}.\lambda \mathtt{x.case\ x\ of\ \ c(y,z) \to c\ (g\ y)\ (f\ g\ z), nil \to nil\ } :: \mathtt{T}}\,(\mathtt{Abs})}{\emptyset; \Delta \vdash \mathtt{M}\ :: (\mathtt{Nat} \longrightarrow \mathtt{Nat}) \longrightarrow [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]}\,(\mathtt{Let})$$

where T is a shorthand notation for $(\mathtt{Nat} \longrightarrow \mathtt{Nat}) \longrightarrow [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]$, where the derivation of the base case nil has been omitted for readability and where the contexts Δ, Γ, Γ' are such that $\Delta(\mathtt{c}) = \mathtt{Nat} \longrightarrow [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]$, $\Delta(\mathtt{nil}) = [\mathtt{Nat}]$ and $\Gamma = \mathtt{f} :: \mathtt{T}, \mathtt{g} :: \mathtt{Nat} \longrightarrow \mathtt{Nat}$ and $\Gamma' = \Gamma, \mathtt{y} :: \mathtt{Nat}, \mathtt{z} :: [\mathtt{Nat}]$. Consequently, the order of M (with respect to T) is equal to 2, as $\mathtt{ord}(\mathtt{T}) = 2$.

3. Interpretations

3.1. **Interpretations of types.** We briefly recall some basic definitions that are very close from the notions used in denotational semantics (See [Win93]) since, as we shall see later, interpretations can be defined in terms of 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 $\top$ is the greatest element satisfying $\forall n \in \overline{\mathbb{N}},\ n \leq \top, n \sqcup \top = \top \sqcup n = \top$ and $n \sqcap \top = \top \sqcap n = n$.

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

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

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

$$\begin{aligned} n \leq_{\overline{\mathbb{N}}} m \text{ iff } n \leq m \\ F \leq_{A \longrightarrow^{\uparrow} B} G \text{ iff } \forall X \in A,\ F(X) \leq_B G(X) \end{aligned}$$

Example 3.1. The type $\mathtt{T} = (\mathtt{Nat} \longrightarrow \mathtt{Nat}) \longrightarrow [\mathtt{Nat}] \longrightarrow [\mathtt{Nat}]$ of the term letRec f = M in Example 2.3 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 $\overrightarrow{F}$ of m terms in the interpretation domain and a sequence $\overrightarrow{\mathtt{T}}$ of k types, the notation $\overrightarrow{F} \in \overrightarrow{(\!|\mathtt{T}|\!)}$ means that both $k = m$ and $\forall i \in [1, m],\ F_i \in (\!|\mathtt{T}_i|\!)$.

3.2. **Interpretations of terms.** Each closed term of type $\mathtt{T}$ will be interpreted by a function 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 of the interpretation domain will be denoted using upper case letters. When needed, Church typing discipline will be used in order to highlight the type of the bound variable in a lambda abstraction.

An important distinction between the terms of the 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 order to obtain complete lattices, each type $(\!|\mathtt{T}|\!)$ has to be completed by a lower bound $\bot_{(\!|\mathtt{T}|\!)}$ and an upper bound $\top_{(\!|\mathtt{T}|\!)}$ as follows:

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

Lemma 3.2. *For each* $\mathtt{T}$ *and for each* $F \in (\!|\mathtt{T}|\!)$, $\bot_{(\!|\mathtt{T}|\!)} \leq_{(\!|\mathtt{T}|\!)} F \leq_{(\!|\mathtt{T}|\!)} \top_{(\!|\mathtt{T}|\!)}$.

Proof. By induction on types. □

Notice that for each type $\mathtt{T}$ it also holds that $\top_{(\!|\mathtt{T}|\!)} \leq_{(\!|\mathtt{T}|\!)} \top_{(\!|\mathtt{T}|\!)}$, by an easy induction.

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

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

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. Moreover, given a boolean predicate P on functions, we will use the notation $\sqcup_P\{F\}$ as a shorthand notation for $\sqcup\{F \mid P\}$.

Lemma 3.3. *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 $\forall F \in S$, $\forall X$, $F(X) \leq G(X)$. 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 $\mathtt{T}$. 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:

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

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

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

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.

For each type $(\!|\mathtt{T}|\!)$, we can define a strict ordering $<$ by: $F < G$ whenever $1 \oplus F \leq G$.

Definition 3.4. 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 $(\!|\mathtt{T}|\!)$.

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

Definition 3.5 (Interpretation)**.** 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 by:

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

where $(\!|\mathtt{op}|\!)_\rho$ is a non decreasing total function such that:

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

The aim of the interpretation of a term is to give a bound on its computation time as we will shortly see in Corollary 3.11. For that purpose, it requires a strict decrease of the interpretation under $\Rightarrow$. This is the object of Lemma 3.9. This is the reason why any *"construct"* of the language involves a $1\oplus$ in each rule of Definition 3.5. Application that plays the role of a *"destructor"* does not require this. This is also the reason why the interpretation of a constructor symbol does not depend on its nature.

Remark that the condition on the the interpretation of operators correspond to the notion of sup-interpretation (See [Péc13] for more details).

3.3. **Existence of an interpretation.** The interpretation of a term is always defined. Indeed, in Definition 3.5, $(\!|\mathtt{letRec}\ \mathtt{f} = \mathtt{M}|\!)_\rho$ 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 [Tar55, KS01] Fixpoint Theorem:

Proposition 3.6. *Each term* $\mathtt{M}$ *of type* $\mathtt{T}$ *has an interpretation.*

Proof. By Lemma 3.3, $L = ((\!|\mathtt{T}|\!), \leq, \sqcup, \sqcap, \top, \bot)$ 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 $\mathtt{letRec}$ 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. **Properties of interpretations.** We now show intermediate lemmata. The following Lemma can be shown by structural induction on terms:

Lemma 3.7. *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 3.8. *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.

$$\begin{aligned}
(\!|\lambda\mathtt{x}.\mathtt{M}\ \mathtt{N}|\!)_\rho &= (\!|\lambda\mathtt{x}.\mathtt{M}|\!)_\rho(\!|\mathtt{N}|\!)_\rho && \text{By Definition 3.5}\\
&= (\Lambda(\!|\mathtt{x}|\!)_\rho.1 \oplus (\!|\mathtt{M}|\!)_\rho)\ (\!|\mathtt{N}|\!)_\rho && \text{By Definition 3.5}\\
&= 1 \oplus (\!|\mathtt{M}|\!)_\rho\{(\!|\mathtt{N}|\!)_\rho/(\!|\mathtt{x}|\!)_\rho\} && \text{By definition of } =\\
&= 1 \oplus (\!|\mathtt{M}\{\mathtt{N}/\mathtt{x}\}|\!)_\rho && \text{By Lemma 3.7}\\
&> (\!|\mathtt{M}\{\mathtt{N}/\mathtt{x}\}|\!)_\rho && \text{By definition of } >
\end{aligned}$$

and so the conclusion. □

Lemma 3.9. *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 is not $\to_{\mathtt{op}}$. By Lemma 3.8, in the case of a β-reduction and, by induction, by Lemma 3.7 and Definition 3.5 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:

$$\begin{aligned}
(\!|\mathtt{M}|\!)_\rho &= \sqcap\{F \in (\!|\mathtt{T}|\!) \mid F \geq 1 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)\ F)\}\\
&\geq \sqcap\{1 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)\ F) \mid F \geq 1 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)\ F)\}\\
&\geq 1 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho) \sqcap \{F \mid F \geq 1((\oplus\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)\ F)\})\\
&\geq 1 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\mathtt{N}|\!)_\rho)\ (\!|\mathtt{M}|\!)_\rho)\\
&\geq 1 \oplus (\!|\mathtt{N}|\!)_\rho\{(\!|\mathtt{M}|\!)_\rho/(\!|\mathtt{f}|\!)_\rho\}\\
&\geq 1 \oplus (\!|\mathtt{N}\{\mathtt{M}/\mathtt{f}\}|\!)_\rho, \quad \text{by Lemma 3.7}\\
&> (\!|\mathtt{N}\{\mathtt{M}/\mathtt{f}\}|\!)_\rho, \quad \text{By definition of } >
\end{aligned}$$

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 non decreasing function (as the interpretation domain only consists in such functions). □

As each reduction distinct from an operator evaluation corresponds to a strict decrease, the following corollary can be obtained:

Corollary 3.10. *For all terms,* $\mathtt{M}$, $\vec{\mathtt{N}}$, *such that* $\emptyset; \Delta \vdash \mathtt{M}\ \vec{\mathtt{N}} :: \mathtt{T}$, *if* $\mathtt{M}\ \vec{\mathtt{N}} \Rightarrow^k \mathtt{M}'$ *then* $(\!|\mathtt{M}|\!)_\rho(\!|\vec{\mathtt{N}}|\!)_\rho \geq |\mathtt{M}\ \vec{\mathtt{N}} \Rightarrow^k \mathtt{M}'| \oplus (\!|\mathtt{M}'|\!)_\rho$.

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

Corollary 3.11. *For all terms,* $\mathtt{M}, \vec{\mathtt{N}}$*, such that* $\emptyset; \Delta \vdash \mathtt{M}\,\vec{\mathtt{N}} :: \mathtt{b}$*, with* $\mathtt{b} \in \boldsymbol{B}$*, if* $(\!|\mathtt{M}\,\vec{\mathtt{N}}|\!)_\rho \neq \top$ *then* $\mathtt{M}\,\vec{\mathtt{N}}$ *terminates in a number of reduction steps in* $O((\!|\mathtt{M}\,\vec{\mathtt{N}}|\!)_\rho)$.

The size $|\mathtt{v}|$ of a value $\mathtt{v}$ (introduced in Subsection 2.2) is defined by $|\mathtt{c}| = 1$ and $|\mathtt{M}\,\mathtt{N}| = |\mathtt{M}| + |\mathtt{N}|$.

Lemma 3.12. *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 3.13. Consider the following term $\mathtt{M} :: \mathtt{Nat} \longrightarrow \mathtt{Nat}$ computing the double of a unary number given as input:

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

We can see below how the interpretation rules of Definition 3.5 are applied on such a term.

$$\begin{aligned}(\!|\mathtt{M}|\!)_\rho &= \sqcap\{F \mid F \geq 1 \oplus ((\Lambda(\!|\mathtt{f}|\!)_\rho.(\!|\lambda\mathtt{x.case\ x\ of\ } +1(\mathtt{y}) \to +1(+1(\mathtt{f\ y}))|\ 0 \to 0|\!)_\rho)F)\}\\
&= \sqcap\{F \mid F \geq 2 \oplus (\Lambda(\!|\mathtt{x}|\!)_\rho.(\!|\mathtt{case\ x\ of\ } +1(\mathtt{y}) \to +1(+1(\mathtt{f\ y}))|\ 0 \to 0|\!)_\rho\{F/(\!|\mathtt{f}|\!)_\rho\})\}\\
&= \sqcap\{F \mid F \geq 3 \oplus (\Lambda X.X \oplus ((\sqcup_{X \geq (\!|+1(\mathtt{y})|\!)_\rho}(\!|+1(+1(\mathtt{f\ y}))|\!)_\rho) \sqcup (\sqcup_{X \geq (\!|0|\!)_\rho}(\!|0|\!)_\rho)\{F/(\!|\mathtt{f}|\!)_\rho\})\}\\
&= \sqcap\{F \mid F \geq 3 \oplus (\Lambda X.X \oplus ((\sqcup_{X \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho} 2 \oplus ((\!|\mathtt{f}|\!)_\rho\ (\!|\mathtt{y}|\!)_\rho)) \sqcup (\sqcup_{X \geq 1} 1)\{F/(\!|\mathtt{f}|\!)_\rho\})\}\\
&= \sqcap\{F \mid F \geq 3 \oplus (\Lambda X.X \oplus ((\sqcup_{X \geq 1 \oplus (\!|\mathtt{y}|\!)_\rho} 2 \oplus (F\ (\!|\mathtt{y}|\!)_\rho)) \sqcup (1))\}\\
&= \sqcap\{F \mid F \geq 3 \oplus (\Lambda X.X \oplus ((2 \oplus (F\ (X-1))) \sqcup (1))\}, \quad X - 1 \geq 0\\
&= \sqcap\{F \mid F \geq \Lambda X.(5 \oplus X \oplus (F\ (X-1))) \sqcup (4 \oplus X)\}\\
&\leq \Lambda X.6X^2 + 5\end{aligned}$$

At the end, we search for the minimal non decreasing function F greater than $\Lambda X.(5 \oplus X \oplus (F\ (X-1))) \sqcup (4 \oplus X)$, for $X > 1$. As the function $\Lambda X.6X^2 \oplus 5$ is a solution of such an inequality, the fixpoint is smaller than this function. Notice that such an interpretation is not tight as one should have expected the interpretation of such a program to be $\Lambda X.2X$. This interpretation underlies that:

- the iteration steps, distinct from the base case, count for $5 \oplus X$: 1 for the `letRec` call, 1 for the application, $1 \oplus X$ for pattern matching and 2 for the extra-constructors added,
- the base case counts for $4 \oplus X$: 1 for recursive call, 1 for application, $1 \oplus X$ for pattern matching and 1 for the 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 first-order interpretation based methods.

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 constraints 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 3.14. *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 trivial interpretation $\top_{(\!|\mathtt{T}|\!)}$. Consequently, these approximations have to be performed with moderation as taking too big intermediate upper bounds might lead to an uninteresting upper bound on the interpretation of the term.

Example 3.15. Consider the term `M` of Example 2.3:

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

Its interpretation $(\!|\mathtt{M}|\!)_\rho$ is equal to:

$$\begin{aligned}
&= \sqcap\{F \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.(\!|\mathtt{x}|\!)_\rho \oplus (\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.(\!|\mathtt{x}|\!)_\rho \oplus (\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.(\!|\mathtt{x}|\!)_\rho \oplus (\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{x}|\!)_\rho \oplus ((((\!|\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 (2 \times (\!|\mathtt{x}|\!)_\rho)^2
\end{aligned}$$

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 a first step of relaxation where we find an upper bound for the max. The below inequality holds for any non decreasing function F:

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

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 $F = \Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(5 \oplus ((\!|\mathtt{g}|\!)_\rho\ (\!|\mathtt{x}|\!)_\rho)) \times (2 \times (\!|\mathtt{x}|\!)_\rho)^2$, where $\times$ is the usual multiplication symbol over natural numbers, satisfies the inequality:

$$F \geq 5 \oplus (\Lambda(\!|\mathtt{g}|\!)_\rho.\Lambda(\!|\mathtt{x}|\!)_\rho.(\!|\mathtt{x}|\!)_\rho \oplus ((((\!|\mathtt{g}|\!)_\rho\ ((\!|\mathtt{x}|\!)_\rho - 1)) + (F\ (\!|\mathtt{g}|\!)_\rho\ ((\!|\mathtt{x}|\!)_\rho - 1))))).$$

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 to higher order polynomials which are the higher order counterpart to polynomials in this theory of complexity.

Definition 3.16. We consider types built from the basic type $\overline{\mathbb{N}}$ as follows:

$$A, B ::= \overline{\mathbb{N}} \mid A \longrightarrow B.$$

Higher order polynomials are built by the following grammar:

$$P, Q ::= c^{\overline{\mathbb{N}}} \mid X^A \mid +^{\overline{\mathbb{N}}\to\overline{\mathbb{N}}\to\overline{\mathbb{N}}} \mid \times^{\overline{\mathbb{N}}\to\overline{\mathbb{N}}\to\overline{\mathbb{N}}} \mid (P^{A\to B}\, Q^A)^B \mid (\Lambda X^A.P^B)^{A\to B}.$$

where c represents constants in $\mathbb{N}$ and P^A means that P is of type A. A polynomial P^A is of order i if $\mathtt{ord}(A) = i$. When A is explicit from the context, we use the notation P_i to denote a polynomial of order i.

In the above definition, constants of type $\overline{\mathbb{N}}$ are distinct from $\mathtt{T}$. By definition, a higher order polynomial P_i has arguments of order at most $i - 1$. For notational convenience, we will use the application of $+$ and $\times$ with an infix notation as in the following example.

Example 3.17. Here are several examples of polynomials generated by the grammar of Definition 3.16:

- $P_1 = \Lambda X_0.(6 \times X_0^2 + 5)$ is an order 1 polynomial,
- $Q_1 = \times$ is an order 1 polynomial,
- $P_2 = \Lambda X_1.\Lambda X_0.(3 \times (X_1\ (6 \times X_0^2 + 5)) + X_0)$ is an order 2 polynomial,
- $Q_2 = \Lambda X_1.\Lambda X_0.((X_1\ (X_1\ 4)) + (X_1\ X_0))$ is an order 2 polynomial.

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

Definition 3.18. Let FP_i, $i \geq 0$, be the class of polynomial functions at order i that consist in functions computed by a term $\emptyset; \Delta \vdash \mathtt{M} :: \mathtt{T}$ over the basic type $\mathtt{Nat}$ and such that:

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

Example 3.19. The term $\mathtt{M}$ of Example 3.15 has order 1 and admits an interpretation bounded by $P_1 = \Lambda X_0.6X_0^2 + 5$. Consequently, $[\![\mathtt{M}]\!] \in \mathrm{FP}_1$.

4. Basic Feasible Functionals

The class of tractable type 2 functionals has been introduced by Constable and Mehlhorn [Con73, Meh74]. It was later named BFF for the class of Basic Feasible Functionals and characterized in terms of function algebra [CK89, IRK01]. We choose to define the class through a characterization by Bounded Typed Loop Programs from [IKR02] which extends the original BFF to any order.

4.1. **Bounded Typed Loop Programs.**

Definition 4.1 (BTLP)**.** A Bounded Typed Loop Program (BTLP) is a non-recursive and well-formed procedure defined by the grammar of Figure 3.

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. 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

$$
\begin{array}{rl}
\text{(Procedures)} \ni P & ::= \textbf{Procedure } v^{\tau_1\times\ldots\times\tau_n\to \mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})\ P^*\ V\ I^*\ \textbf{Return } v_r^{\mathtt{D}} \\
\text{(Declarations)} \ni V & ::= \textbf{Var } v_1^{\mathtt{D}},\ldots,v_n^{\mathtt{D}}; \\
\text{(Instructions)} \ni I & ::= v^{\mathtt{D}} := E;\ |\ \textbf{Loop } v_0^{\mathtt{D}} \textbf{ with } v_1^{\mathtt{D}} \textbf{ do } \{I^*\} \\
\text{(Expressions)} \ni E & ::= 0 \mid 1 \mid v^{\mathtt{D}} \mid v_0^{\mathtt{D}} + v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}} - v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}} \# v_1^{\mathtt{D}} \mid v^{\tau_1\times\ldots\times\tau_n\to \mathtt{D}}(A_1^{\tau_1},\ldots,A_n^{\tau_n}) \\
\text{(Arguments)} \ni A & ::= v \mid \lambda v_1,\ldots,v_n.v(v'_1\ldots,v'_m) \quad \text{with } v\notin\{v_1,\ldots,v_n\}
\end{array}
$$

Figure 3: BTLP grammar

```
Procedure SumUp(F^((D→D)→D), x^D)
    Var bnd, maxi, sum, i;
    maxi := 0;
    i := 0;
    Loop x with x do {
        If F(λz.maxi) < F(λz.i) {
            maxi := i;
        }
        i := i+1;
    }
    bnd := (x+1)#(F(λz.maxi)+1);
    sum := 0;
    i := 0;
    Loop x with bnd do {
        sum := sum + F(λz.i);
        i := i+1;
    }
    Return sum^D
```

Figure 4: Example BTLP procedure

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.

A BTLP procedure $\textbf{Procedure } v^{\tau_1\times\ldots\times\tau_n\to \mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})\ P^*\ V\ I^*\ \textbf{Return } v_r^{\mathtt{D}}$ computes an order i functional if and only if $\mathtt{ord}(\tau_1\times\ldots\times\tau_n\to\mathtt{D}) = i$. The order of BTLP types can be defined similarly to the order on types of Definition 2.2, extended by $\mathtt{ord}(\tau_1\times\ldots\times\tau_n) = \max_{i=1}^{n}(\mathtt{ord}(\tau_i))$ and where $\mathtt{D}$ is considered to be a basic type.

An example BTLP procedure is provided in Figure 4. This example procedure SumUp is directly sourced from [IKR02] and is an order 3 procedure that computes the function: $F, x \mapsto \sum_{i<|x|} F(\lambda z.i)$. It first computes a bound `bnd` on the result by finding the number `i` for which $F(\lambda z.i)$ is maximal and then computes the sum itself. Note that this procedure uses a conditional statement (**If**) not

included in the grammar but that can be simulated in BTLP (see [IKR02]). Such a conditional will be explicitly added to the syntax of IBTLP procedures in Definition 5.3.

Let the size of an argument be the number of syntactic elements in it. The size of input arguments is the sum of the size of the arguments in the input.

Definition 4.2 (Time complexity)**.** For a given procedure P of parameters $(v_1^{\tau_1}, \ldots, v_n^{\tau_n})$, we define its time complexity $t(P)$ to be a function of type $\mathbb{N} \to \mathbb{N}$ that, given an input of type $\tau_1 \times \ldots \times \tau_n$ returns the maximal number of assignments executed during the evaluation of the procedure in the size of the input.

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

Definition 4.3 (BFF_i)**.** For any $i \geq 1$, BFF_i is the class of order i functionals computable by a BTLP procedure[2].

It was demonstrated in [IRK01] that $\mathrm{BFF}_1 = \mathrm{FPTIME}$ and BFF_2=BFF.

4.2. **Safe Feasible Functionals.** Now we restrict the domain of BFF_i classes to inputs in BFF_k for $k < i$, the obtained classes are named SFF for Safe Feasible Functionals.

Definition 4.4 (SFF_i)**.** SFF_1 is defined to be the class of order 1 functionals computable by BTLP a procedure and, for any $i \geq 1$, SFF_{i+1} is the class of order $i+1$ functionals computable by BTLP a procedure on the input domain SFF_i. In other words,

$$\begin{aligned} \mathrm{SFF}_1 &= \mathrm{BFF}_1, \\ \mathrm{SFF}_{i+1} &= \mathrm{BFF}_{i+1}{}_{\restriction \mathrm{SFF}_i} \qquad\qquad \forall i,\ i \geq 1 \end{aligned}$$

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 $k < i$. 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

In this section, we show our main characterization of safe feasible functionals:

Theorem 5.1. *For any order $i \geq 0$, the class of functions in FP_{i+1} over FP_k, $k \leq i$, is exactly the class of functions in SFF_{i+1}, up to an isomorphism. In other words,*

$$\mathrm{SFF}_{i+1} \equiv \mathrm{FP}_{i+1\restriction(\cup_{k\leq i}\mathrm{FP}_k)},$$

for all $i \geq 0$, up to an isomorphism.

Proof. For a fixed i, the theorem is proved in two steps: Soundness, Theorem 5.2, and Completeness, Theorem 5.14. Soundness consists in showing that any term M whose interpretation is bounded by an order i polynomial, computes a function in SFF_i. Completeness consists in showing that any BTLP procedure P of order i can be encoded by a term M computing the same function and admitting a polynomial interpretation of order i. □

[2]As demonstrated in [IKR02], all types in the procedure can be restricted to be of order at most i without any distinction.

Notice that functions in SFF_{i+1} return the dyadic representation of a natural number. Consequently, the isomorphism is used on functions in FP_i to illustrate that a function of type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ and order 1 is isomorphic to a functional of type $\mathbb{N} \to \mathbb{N}$ and of the same order using decurryfication and pair encoding over $\mathbb{N}$. In order to simplify the treatment we will restrict ourselves to *functional terms* computing functionals that are terms of type $\mathtt{T} \longrightarrow \mathtt{b}$, with $\mathtt{b} \in \mathbf{B}$, in the remaining of the paper.

5.1. **Soundness.** The soundness means that any term whose interpretation is bounded by an order i polynomial belongs to SFF_i. For that, we will show that the interpretation allows us to bound the computing time with an higher order polynomial.

Theorem 5.2. *Any functional term* $\mathtt{M}$ *whose interpretation is bounded by an order* i *polynomial, computes a functional in* SFF_i.

Proof. For order 1, consider that the term $\mathtt{M}$ has an interpretation bounded by a polynomial P_1. For any value $\mathtt{v}$, we have, by Corollary 3.10, that the computing time of $\mathtt{M}$ on input $\mathtt{v}$ bounded by $(\!|\mathtt{M}\ \mathtt{v}|\!)_\rho$. Consequently, using Lemma 3.12, we have that:

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

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

Now, for higher order, let $\mathtt{M}$ be an order $i+1$ term of interpretation $(\!|\mathtt{M}|\!)_\rho$. There exists an order $i+1$ polynomial P_{i+1} such that $(\!|\mathtt{M}|\!)_\rho \leq P_{i+1}$. We know that on input $\mathtt{N}$, $\mathtt{M}$ normalizes in $O((\!|\mathtt{M}\ \mathtt{N}|\!)_\rho)$, by Corollary 3.11. Since $\mathtt{N}$ computes a functional $[\![\mathtt{N}]\!] \in \mathrm{SFF}_i$ there is a polynomial P_i such that $(\!|\mathtt{N}|\!)_\rho \leq P_i$, by induction on the order 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 $Q_{i+1} = P_{i+1} \circ P_i$. □

The above result holds for terms over arbitrary basic inductive types, by considering that each value on such a type encodes an integer value.

5.2. **Completeness.** To prove that all functions computable by a BTLP program of order i can be defined as terms admitting a polynomial interpretation, we proceed in several steps:

(1) We show that it is possible to encode each BTLP procedure P into an intermediate procedure $\langle P \rangle$ of a language called IBTLP (See Figure 5) for If-Then-Else Bounded Typed Loop Program such that $\langle P \rangle$ computes the same function as P using the same number of assignments (*i.e.* with the same time complexity).
(2) We show that we can translate each IBTLP procedure $\langle P \rangle$ into a flat IBTLP procedure $\overline{\langle P \rangle}$, *i.e.* a procedure with no nested loops and no procedure calls.
(3) Then we transform the flat IBTLP procedure $\overline{\langle P \rangle}$ into a "local" and flat IBTLP procedure $[\overline{\langle P \rangle}]_\emptyset$ checking bounds locally in each assignment instead of checking it globally in each loop. For that purpose, we use a polynomial time computable operator of the IBTLP language called `chkbd`. The time complexity is then asymptotically preserved.
(4) Finally, we compile the IBTLP procedure $[\overline{\langle P \rangle}]_\emptyset$ into a term of our language and we use completeness for first order function to show that there is a term computing the same function and admitting a polynomial interpretation. This latter transformation does not change the program behavior in terms of computability and complexity, up to a O, but it makes the simulation by a functional program easier as each local assignment can be simulated independently of the context.

$$
\begin{aligned}
IP &::= v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})\{IP^*\,IV\,II^*\}\ \textbf{ret}\ v_r^{\mathtt{D}}\\
IV &::= \textbf{var}\ v_1^{\mathtt{D}},\ldots,v_n^{\mathtt{D}};\\
II &::= v^{\mathtt{D}}\ :=\ IE^{\mathtt{D}};\ \mid \textbf{loop}\ v_0^{\mathtt{D}}\ \{II^*\} \mid \textbf{if}\ v^{\mathtt{D}}\ \{\ II^*\}\ [\textbf{else}\ \{\ II^*\}] \mid (II^*)_{v^{\mathtt{D}}}\\
IE &::= 0 \mid 1 \mid v^{\mathtt{D}} \mid v_0^{\mathtt{D}}+v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}}-v_1^{\mathtt{D}} \mid v_0^{\mathtt{D}}\#v_1^{\mathtt{D}} \mid v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(IA_1^{\tau_1},\ldots,IA_n^{\tau_n}) \mid v_0^{\mathtt{D}}\times v_1^{\mathtt{D}} \mid \mathtt{cut}(v^{\mathtt{D}})\\
&\quad \mid \mathtt{chkbd}(E,X)\\
IA &::= v \mid \lambda v_1,\ldots,v_n.v(v_1'\ldots,v_m') \quad \text{with } v\notin\{v_1,\ldots,v_n\}
\end{aligned}
$$

Figure 5: IBTLP grammar

The 3 first steps just consist in transforming a BTLP procedure into a IBTLP procedure in order to simplify the compilation procedure of the last step. These steps can be subsumed as follows:

$$
\begin{array}{lccccccc}
\text{Language} & \text{BTLP} & & \text{IBTLP} & & \text{IBTLP} & & \text{IBTLP} \\
\text{Program} & P & \xrightarrow{()} & (P) & \xrightarrow{\text{flat}} & \overline{(P)} & \xrightarrow{\text{local}} & [\overline{(P)}]_\emptyset \xrightarrow{\text{compile}} \mathtt{comp}([\overline{(P)}]_\emptyset)
\end{array}
$$

For each step, we check that the complexity in terms of reduction steps is preserved and that the transformed program computes the same function.

5.2.1. *From BTLP to IBTLP.*

Definition 5.3 (IBTLP)**.** A If-Then-Else Bounded Typed Loop Program (IBTLP) is a non-recursive and well-formed procedure defined by the grammar of Figure 5.

The well-formedness assumption and variable bounds are the same as for BTLP. In a loop statement, the guard variable v_0 still cannot be assigned to within II^*. A IBTLP procedure IP has also a time complexity $t(IP)$ defined similarly to the one of BTLP procedures.

The main distinctions between an IBTLP procedure and a BTLP procedure are the following:

- there are no loop bounds in IBTLP loops. Instead loop bounds are written as instruction annotations: a IBTPL loop $(\textbf{loop}\ v_0^{\mathtt{D}}\ \{II^*\})_{v^{\mathtt{D}}}$ corresponds to a BTLP instruction $\textbf{Loop}\ v_0^{\mathtt{D}}\ \textbf{with}\ v^{\mathtt{D}}\ \textbf{do}\ \{II^*\}$.
- IBTLP includes a conditional statement $\textbf{if}\ v^{\mathtt{D}}\ \{\ II_1^*\ \}\ \textbf{else}\ \{\ II_2^*\}$ evaluated in a standard way: if variable v is 0 then it is evaluated to II_2^*. In all other cases, it is evaluated to II_1^*, the else branching being optional.
- IBTLP includes a basic operator $\times$ such that $x\times y = 2^{|x|+|y|}$.
- IBTLP includes a unary operator `cut` which removes the first character of a number (*i.e.* $\mathtt{cut}(0) = 0$, $\mathtt{cut}(2x+i) = x$ where $i\in\{0,1\}$).
- IBTLP includes an operator `chkbd` computing the following function:

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

  where $[\![E]\!]$ is the dyadic number obtained after the evaluation of expression E and X is a finite set of variables.

Notice that `chkbd` is in SFF_1 provided that the input E is computable in polynomial time and both $\times$ and `cut` are in SFF_1. The semantics of an IBTLP procedure is also similar to the one of a BTLP procedure: during the execution of an assignment, the bound check is performed on instruction

annotations instead of being performed on loop bounds. However IBTLP is strictly more expressive than BTLP from an extensional perspective: a loop can be unbounded. This is the reason why only IBTLP procedures obtained by well-behaved transformation from BTLP procedures will be considered in the remainder of the paper.

Now we define a program transformation $(.)$ from BTLP to IBTLP. For each loop of a BTLP program, this transformation just consists in recording the variable appearing in the **with** argument of a contextual loop and putting it into an instruction annotation as follows:

$$(\textbf{Loop } v_0^{\mathbb{D}} \textbf{ with } v_1^{\mathbb{D}} \textbf{ do } \{I^*\}) = (\textbf{loop } v_0^{\mathbb{D}} \{(I)^*\})_{v_1^{\mathbb{D}}}$$

Any assignment is left unchanged and this transformation is propagated inductively on procedure instructions so that any inner loop is transformed. We denote by (P) the IBTLP procedure obtained from the BTLP procedure P. Hence the following lemma straightforwardly holds:

Lemma 5.4. *Given a procedure P, let $[\![P]\!]$ denote the function computed by P. For any BTLP procedure P, we have $[\![P]\!] = [\![(P)]\!]$ and $t(P) = t((P))$.*

Proof. The transformation is semantics-preserving (the computed function is the same). Any assignment in P corresponds to exactly one assignment in (P) and the number of iterations of **Loop** v **with** … and **loop** v are both in $|v|$. □

5.2.2. *From IBTLP to Flat IBTLP.*

Definition 5.5 (Flat IBTLP)**.** A If-Then-Else Bounded Typed Loop Program (IBTLP) is *flat* if it does not contain nested loops.

We will show that it is possible to translate any IBTLP procedure into a Flat IBTLP procedure using the `if` construct.

There are only three patterns of transformation:

(1) one pattern for nested loops, called *Unnest*,
(2) one pattern for sequential loops, called *Unseq*,
(3) and one for procedure calls inside a loop, called *Unfold*

that we describe below:

(1) The first transformation *Unnest* consists in removing a nested loop of a given procedure. Assume we have a IBTLP procedure with two nested loops:

`(loop x {` II_1^* `(loop y {`II_2^*`})`$_z$ II_3^*`})`$_w$

we can translate it to a IBTLP procedure as:

```
total := w×y; dx := 1; gt := total; gy := y;
lb := x#total;
loop lb {
  if dx {(II1*)w}
  if gy {((II2*)z)w gy := cut(gy);}
  if dx {dx := 0; (II3*)w}
  if gt {gt := cut(gt);}
  else {gt := total; gy := y; dx := 1;}
}
```

where all the variables dx, total, gt, gy and lb are fresh local variables and II_1^*, II_2^* and II_3^* are instructions with no loop. The intuition is that variables dx and gy tell whether the loop should execute II_1^* and II_2^* respectively. Variable gt counts the number of times to execute the subloop.

(2) The second transformation Unseq consists in removing two sequential loops of a given procedure. Assume we have a IBTLP procedure with two sequential loops,

$(\texttt{loop x}_1\ \{II_1^*\})_{w_1}\ II_2^*\ (\texttt{loop x}_3\ \{II_3^*\})_{w_3}$

We can translate it to a single loop IBTLP procedure as:

```
gx := x1; dy := 1; lb := x1×x3;
loop lb {
  if gx {gx := cut gx; (II1*)w1}
  else  {
    if dy {II2*}
    else {dy := 0; (II3*)w3}
  }
}
```

where gx, dy and lb are fresh local variables and II_1^*, II_2^* and II_3^* are instructions with no loop.

(3) The last transformation *Unfold* consists in removing one procedure call of a given procedure. Assume we have a IBTLP procedure with a call to procedure P of arity n: $X := P(IA_1, ..., IA_n)$;

We can translate it to a computationally equivalent IBTLP procedure after removing the call to procedure P. For that purpose, we carefully alpha-rename all the variables of the procedure declaration (parameters and local variables) to obtain the procedure $P(v_1, \dots, v_n)\{IP^* IV\, II^*\}$ **ret** v_r then we add the procedure declarations IP^* to the caller procedure list of procedure declarations, and the local variables IV and parameters $v_1, \dots, v_n$ to the caller procedure list of local variable declarations and then we generate the following code:

$v_1 := IA_1;\ \dots\ v_n := IA_n;\ II^*\ X := v_r;$

This program transformation can be extended straightforwardly to the case where the procedure call is performed in a general expression. Notice that unfolding a procedure is necessary as nested loops may appear because of procedure calls.

These three patterns can be iterated on a IBTLP procedure (from top to bottom) to obtain a Flat IBTLP procedure in the following way:

Definition 5.6. The transformation $\overline{IP}$ is a mapping from IBTLP to IBTLP defined by:

$$\overline{IP} = (Unnest^! \circ Unseq^!)^! \circ Unfold^!(IP),$$

where, for given function f, $f^!$ is the least fixpoint of f on a given input and $\circ$ is the usual mapping composition.

Notice that this procedure is polynomial time computable as each application of an Unfold call consumes a procedure call (and procedures are non recursive) and each application of an Unnest or Unseq call consumes one loop. Consequently, fixpoints always exist.

Lemma 5.7. *For any BTLP procedure P, $\overline{(P)}$ is a flat IBTLP procedure.*

Proof. First notice that repeated application of the *Unfold* pattern may only introduce a constant number of new loops as procedures are non-recursive. Consequently, the fixpoint $Unfold^!$ is defined

$$
\begin{aligned}
[v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})IP^*\ IV\ II^*\ \mathbf{ret}\ v_r^{\mathtt{D}}]_X &= v^{\tau_1\times\ldots\times\tau_n\to\mathtt{D}}(v_1^{\tau_1},\ldots,v_n^{\tau_n})IP^*\ IV\ [II]_X^*\ \mathbf{ret}\ v_r\\
[II_v]_X &= [II]_{X\cup\{v\}}\\
[\mathbf{if}\ v^{\mathtt{D}}\ \{\ II_1^*\ \}\ \mathbf{else}\ \{\ II_2^*\}]_X &= \mathbf{if}\ v^{\mathtt{D}}\ \{\ [II_1]_X^*\ \}\ \mathbf{else}\ \{\ [II_2]_X^*\}\\
[\mathbf{loop}\ v_0^{\mathtt{D}}\{II^*\};]_X &= \mathbf{loop}\ v_0^{\mathtt{D}}\ \{[II]_X^*\}\\
[v^{\mathtt{D}}\ :=\ IE;]_X &= v^{\mathtt{D}}\ :=\ \mathtt{chkbd}(IE^{\mathtt{D}},X);
\end{aligned}
$$

Figure 6: From Flat IBTLP to Local and Flat IBTLP

and reached after a constant number (in the size of the procedure) of applications. Each application of a *Unseq* pattern or *Unnest* pattern decreases by one the number of loop within a procedure. Consequently, a fixpoint is reached (modulo α-equivalence) and the only programs for which such patterns cannot apply are programs with a number of loops smaller than 1. □

Lemma 5.8. *For any BTLP procedure P, we have $[\![\overline{(P)}]\!] = [\![P]\!]$ and $t(\overline{(P)}) = O(t(P))$.*

Proof. In the first equality, the computed functions are the same since the program transformation preserves the extensional properties. For the second equality, the general case can be treated by a structural induction on the procedure (P). For simplicity, we consider the case of a procedure (P) only involving n nested loops over guard variables $x_1,\ldots,x_n$ and loop bounds $x_{n+1},\ldots,x_{2n}$, respectively over one single basic instruction (*e.g.* one assignment with no procedure call). With inputs of size m, this procedure will have a worst case complexity of m^n (when the loop bounds are not reached). Consequently, $t(P)(m) = m^n$. This procedure will be transformed into a flat IBTLP procedure using the Unnest transformation $n-1$ times over a variable z containing the result of the computation $x_1\#((x_2\times x_{n+1})\#(\ldots x_3\times x_{n+2})\#(x_n\times x_{2n})\ldots)$ as initial value. Consequently, on an input of size m, we have $t(\overline{(P)})(m) \leq |z| = |x_1|\times(|x_2|+|x_{n+1}|)\times\ldots\times(|x_n|+|x_{2n}|) \leq 2^{n-1}\times m^n = O(m^n)$, as for each i $|x_i| \leq m$ and by definition of the # and $\times$ operators. We conclude using Lemma 5.4. □

5.2.3. *From Flat IBTLP to Local and Flat IBTLP.* Now we describe the last program transformation $[-]_X$ that makes the check bound performed on instruction annotations explicit. For that purpose, $[-]_X$ makes use of the operator `chkbd` and records a set of variables X (the annotations enclosing the considered instruction). The procedure is described in Figure 6. Notice that the semantics condition ensuring that no assignment must be performed on a computed value of size greater than the size of the loop bounds for a BTLP procedure or than the size of the loop annotations for a IBTLP procedure has been replaced by a local computation performing exactly the same check using the operator `chkbd`. Consequently, we have:

Lemma 5.9. *For any BTLP procedure P, we have $[\![\overline{[(P)]_\emptyset}]\!] = [\![P]\!]$ and $t(\overline{[(P)]_\emptyset}) = O(t(P))$.*

Proof. We use Lemma 5.8 together with the fact that extensionality and complexity are both preserved. □

$$\texttt{comp}(v(v_1,\dots,v_m)\ IP^*\ \textbf{var}\ v_{m+1},\dots,v_n;\ II^*\ \textbf{ret}\ v_r) = \lambda s.\pi^n_r(\texttt{comi}^n(II^*)\ s)$$

$$\texttt{comi}^n(II_1\ \dots\ II_k) = \texttt{comi}^n(II_k)\ \dots\ \texttt{comi}^n(II_1)$$

$$\texttt{comi}^n(v_i := IE;) = \lambda\langle v_1,\dots,v_n\rangle.\langle v_1,\dots,v_{i-1},\texttt{come}(IE),v_{i+1},\dots,v_n\rangle$$

$$\texttt{comi}^n(\textbf{loop}\ v_i\ \{II^*\}) = \lambda\langle v_1,\dots,v_n\rangle.(\left(\begin{array}{rcl}\texttt{letRec}\ f = \lambda\tilde{t}.\lambda s.\texttt{case}\ \tilde{t}\ \texttt{of}\ 0 & \rightarrow & s\\ & & j(t) \rightarrow f\ t\ (\texttt{comi}^n(II^*)\ s)\end{array}\right) v_i)$$

$$\texttt{comi}^n(\textbf{if}\ v_i^{\mathtt{D}}\ \{\ II_1^*\ \}\ \textbf{else}\ \{\ II_2^*\}) = \lambda\langle v_1,\dots,v_n\rangle.\texttt{case}\ v_i\ \texttt{of}\ 0 \rightarrow \texttt{comi}^n(II_2^*) \mid j(t) \rightarrow \texttt{comi}^n(II_1^*)$$
with $j \in \{0,1\}$

$$\texttt{come}(c) = c, \quad c \in \{1, v^{\mathtt{D}}\}$$

$$\texttt{come}(v_0^{\mathtt{D}}\ \texttt{op}\ v_1^{\mathtt{D}}) = \texttt{op}\ v_0\ v_1, \quad \texttt{op} \in \{+,-,\#,\times\}$$

$$\texttt{come}(\texttt{cut}(v^{\mathtt{D}})) = \texttt{cut}\ v$$

$$\texttt{come}(\texttt{chkbd}(IE,\{v_{j_1},\dots,vj_r\})) = \texttt{chkbd}\ \texttt{come}(IE)\ (\sqcap^{\mathtt{D}}_r\ v_{j_1}\ \dots\ v_{j_r})$$

$$\texttt{come}(v(IA_1,\dots,IA_n)) = v\ \texttt{come}(IA_1)\ \dots\ \texttt{come}(IA_n)$$

$$\texttt{come}(\lambda v_1,\dots,v_n.v(v'_1\dots,v'_m)) = \lambda v_1.\dots\lambda v_n.(\dots(v\ v'_1)\ \dots\ v'_m)\dots)$$

Figure 7: Compiler from Local and Flat IBTLP to terms

5.2.4. *From Local and Flat IBTLP to terms.* We then encode Flat and Local IBTLP in our functional language. For this, we define a procedure `comp` that will "compile" IBTLP procedures into terms. For that purpose, we suppose that the target term language includes constructors for numbers (0 and 1), a constructor for tuples $\langle\dots\rangle$, all the operators of IBTLP as basic prefix operators (+, -, #, …), min operators $\sqcap^{\mathtt{D}}_n$ computing the min of the sizes of n dyadic numbers and a `chkbd` operator of arity 2 such that $[\![\texttt{chkbd}]\!](\mathtt{M},\mathtt{N}) = [\![\mathtt{M}]\!]$ if $|[\![\mathtt{M}]\!]| \leq |[\![\mathtt{N}]\!]|$ (and 0 otherwise). All these operators are extended to be total functions in the term language: they return 0 on input terms for which they are undefined. Moreover, we also require that the Flat and Local IBTLP procedures given as input are 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 7, defining `comp`, $\texttt{comi}^n$, `come` that respectively compile procedures, instructions and expressions. The `comi` compiling process is indexed by the number of variables in the program n.

For convenience, let $\lambda\langle v_1,\dots,v_n\rangle.$ be a shorthand notation for $\lambda s.\texttt{case}\ s\ \texttt{of}\ \langle v_1,\dots,v_n\rangle \rightarrow$ and let π^n_r be a shorthand notation for the r-th projection $\lambda\langle v_1,\dots,v_n\rangle.v_r$.

The compilation procedure works as follows: any Local and Flat IBTLP procedure of the shape $v(v_1,\dots,v_m)\ IP^*\ \textbf{var}\ v_{m+1},\dots,v_n;\ II^*\ \textbf{ret}\ v_r$ will be transformed into a term of type $\tau_1\times\dots\times\tau_n\rightarrow\tau_r$, provided that τ_i is the type of v_i and that $\tau_1\times\dots\times\tau_n$ is the type for n-ary tuples of the shape $\langle v_1,\dots,v_n\rangle$. Each instruction within a procedure of type $\tau_1\times\dots\times\tau_n\rightarrow\tau_r$ will have type $\tau_1\times\dots\times\tau_n\rightarrow\tau_1\times\dots\times\tau_n$. Consequently, two sequential instructions $II_1\ II_2$ within a procedure of n variables will be compiled into a term application of the shape $\texttt{comi}^n(II_2)\ \texttt{comi}^n(II_1)$ and instruction type is preserved by composition. Each assignment of the shape $v_i := IE;$ is compiled into a term that takes a tuple as input and returns the identity but on the i-th component. The i-th component is replaced by the term obtained after compilation of IE on which a checkbound is performed. The min operator applied for this checkbound is $\sqcap^{\mathtt{D}}_n$ whenever X is of cardinality n. The compilation process for expressions is quite standard: each construct is replaced by the corresponding construct in the target language. Notice that the compiling procedure is very simple for procedures as it only applies to Flat IBTLP procedures on which any procedure call has been removed by unfolding. The only difficulty

to face is for loop compilation: we make use of a `letRec` of type $\mathtt{D} \to \tau_1 \times \ldots \times \tau_n \to \tau_1 \times \ldots \times \tau_n$. 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 $\tau_1 \times \ldots \times \tau_n \to \tau_1 \times \ldots \times \tau_n$.

Again, for a given term $\mathtt{M}$ of type $\tau_1 \ldots \longrightarrow \ldots \tau_n \longrightarrow \tau$, we define its time complexity $t(\mathtt{M})$ to be a function of type $\mathbb{N} \to \mathbb{N}$ that, for any inputs of type τ_i $\vec{\mathtt{N}}$ and size bounded by m returns the maximmal value of $|\mathtt{M}\,\vec{\mathtt{N}} \Rightarrow [\![\mathtt{M}\,\vec{\mathtt{N}}]\!]|$. In other words, $t(\mathtt{M})(m) = \max_{|\vec{\mathtt{N}}| \leq m}\{|\mathtt{M}\,\vec{\mathtt{N}} \Rightarrow [\![\mathtt{M}\,\vec{\mathtt{N}}]\!]|\}$.

Lemma 5.10. *For any BTLP procedure P, we have $[\![\mathtt{comp}([\overline{(P)}]_\emptyset)]\!] = [\![P]\!]$ and $t(\mathtt{comp}([\overline{(P)}]_\emptyset)) = O(t(P))$.*

Proof. The term obtained by the compilation process is designed to compute the same function as the one computed by the initial procedure. It remains to check that for a given instruction the number of reductions remains in $O(t(P))$. This is clearly the case for an assignment as it consists in performing one beta-reduction, one case reduction, and the evaluation of a fixed number of symbols within the expression to evaluate. For an iteration the complexity of the `letRec` call is in $|v_i|$ as for an IBTLP loop. Indeed each recursive call consists in removing one symbol. Finally, it remains to apply lemma 5.9. □

Example 5.11. Let us take a simple BTLP procedure:

```
Procedure mult(x,y)
Var z,b; z := 0; b := x#y;
Loop x with b do{z := z+y;}
Return z
```

It will be translated in Local and Flat IBTLP as:

```
mult(x,y){
     var z,b; z := 0; b := x#y;
     loop x {z := chkbd(z+y,{b});}
     ret z}
```

And be compiled modulo α-equivalence in a term as:

$$\lambda s.\pi_3^4\left(\begin{array}{l}\lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\left(\mathtt{letRec}\ f = \begin{array}{rcl}\lambda\tilde{t}.\lambda\tilde{s}.\mathtt{case}\ \tilde{t}\ \mathtt{of}\ 0 & \to & s \\ \qquad\qquad\quad j(t) & \to & f\ t\ (\mathtt{M}\ \tilde{s}))\end{array}\right)\mathtt{x} \\ (\lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\langle \mathtt{x},\mathtt{y},\mathtt{z},\#\ \mathtt{x}\ \mathtt{y}\rangle(\lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\langle \mathtt{x},\mathtt{y},0,b\rangle\ s))\end{array}\right)$$

where $\mathtt{M} = \lambda\langle \mathtt{x},\mathtt{y},\mathtt{z},\mathtt{b}\rangle.\langle \mathtt{x},\mathtt{y},\mathtt{chkbd}\ (+\ \mathtt{z}\ \mathtt{y})\ (\sqcap_1^{\mathtt{D}}\ \mathtt{b}),\mathtt{b}\rangle$

Now it remains to check that any term obtained during the compilation procedure has a polynomial interpretation. In order to show this result, we first demonstrate some intermediate results:

Lemma 5.12. *Given an assignment ρ, the operators $+$, $-$, $\#$, $\times$, `cut`, $\sqcap_n^{\mathtt{D}}$ and `chkbd` admit a polynomial sup-interpretation.*

Proof. We can check that the following are polynomial sup-interpretations.

$$\begin{aligned}
(\!|+|\!)_\rho &= \Lambda X.\Lambda Y.(X+Y),\\
(\!|-|\!)_\rho &= \Lambda X.\Lambda Y.X,\\
(\!|\#|\!)_\rho &= \Lambda X.\Lambda Y.(X\times Y),\\
(\!|\times|\!)_\rho &= \Lambda X.\Lambda Y.(X+Y),\\
(\!|\mathtt{cut}|\!)_\rho &= \Lambda X.\sqcup(X-1,0),\\
(\!|\sqcap^{\mathtt{D}}_n|\!)_\rho &= \Lambda X_1.\ldots.\Lambda X_n.\,\overline{\sqcap^{\mathbb{N}}}(X_1,\ldots,X_n),\\
(\!|\mathtt{chkbd}|\!)_\rho &= \Lambda X.\Lambda Y.Y.
\end{aligned}$$

The inequalities are straightforward for the basic operators $+,-,\#,\ldots$. For $(\!|\mathtt{chkbd}|\!)_\rho$, to be a sup-interpretation, we have to check that:

$$\forall \mathtt{M},\forall \mathtt{N},\ (\!|\mathtt{chkbd}\ \mathtt{M}\ \mathtt{N}|\!)_\rho \geq (\!|[\![\mathtt{chkbd}]\!](\mathtt{M},\mathtt{N})|\!)_\rho.$$

$$\begin{aligned}
(\!|\mathtt{chkbd}\ \mathtt{M}\ \mathtt{N}|\!)_\rho &= (\!|\mathtt{chkbd}|\!)_\rho\ (\!|\mathtt{M}|\!)_\rho\ (\!|\mathtt{N}|\!)_\rho\\
&= (\Lambda X.\Lambda Y.Y)\ (\!|\mathtt{M}|\!)_\rho\ (\!|\mathtt{N}|\!)_\rho\\
&= (\!|\mathtt{N}|\!)_\rho\\
&\geq (\!|[\![\mathtt{N}]\!]|\!)_\rho && \text{by Corollary 3.10}\\
&\geq |[\![\mathtt{N}]\!]| && \text{by Lemma 3.12}\\
&\geq |[\![\mathtt{chkbd}]\!](\mathtt{M},\mathtt{N})| && \text{by Definition of } \mathtt{chkbd}\\
&\geq (\!|[\![\mathtt{chkbd}]\!](\mathtt{M},\mathtt{N})|\!)_\rho && \text{by Lemma 3.12}
\end{aligned}$$

and so the conclusion.

In the particular case where $[\![\mathtt{N}]\!]$ is not a value, we have $(\!|[\![\mathtt{chkbd}]\!](\mathtt{M},\mathtt{N})|\!)_\rho = (\!|0|\!)_\rho = 1$ so the inequality is preserved. □

Now we are able to provide the interpretation of each term encoding an expression:

Corollary 5.13. *Given a BTLP procedure P, any term $\mathtt{M}$ obtained by compiling an expression IE (i.e. $\mathtt{M} = \mathtt{come}(IE)$) of the flat and local IBTLP procedure $[\,\overline{(P)}\,]_\emptyset$ admits a polynomial interpretation $(\!|\mathtt{M}|\!)_\rho$.*

Proof. By unfolding, there is no procedure call in the expressions of procedure $[\,\overline{(P)}\,]_\emptyset$. By Lemma 5.12 and by Definition 3.5 , any symbol of an expression has a polynomial interpretation, obtained by finite composition of polynomials. □

Theorem 5.14. *Any BTLP procedure P can be encoded by a term $\mathtt{M}$ computing the same function and admitting a polynomial interpretation.*

Proof. We have demonstrated that any loop can be encoded by a first order term whose runtime is polynomial in the input size. Each higher order expression in a tuple can be encoded by a first order term using defunctionalization. Consequently, by completeness of first-order polynomial interpretations there exists a term computing the same function and admitting a polynomial interpretation. □

Example 5.15. We provide a last example to illustrate the expressive power of the presented methodology. Define the term $\mathtt{M}$ by:

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

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 $\mathtt{M}\ (\lambda \mathtt{x}.1 + (1 + (\mathtt{x}))$ 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 a polynomial interpretation when applied as first operand of this fold function.

6. Conclusion and future work

This paper has introduced a theory for higher-order interpretations that can be used to deal with higher-order complexity classes. We manage to provide a characterization of the complexity classes BFF_i introduced in [IKR02] for any order i, with a restriction on their input. For $i = 1$, we obtain a characterization of FPTIME and, for $i = 2$, we obtain a characterization of BFF applied to FPTIME inputs.

This is a novel approach but there are still some important issues to discuss.

- Synthesis: it is 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 polynomial interpretations over natural numbers (see [Péc13] for a survey). As a consequence this problem is also undecidable for higher order. Some simplification such as defunctionalization and the use of interpretations over the reals can vanish this undecidability trouble.
- Intensionality: The expressive power of interpretations in terms of captured algorithms (also called intensionality) 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.* their interpretation will sometimes be $\top$, although the function will be computed by another algorithm (or term) admitting a finite interpretation. At least, the presented paper has shown that the expressive power of interpretations can be extended to higher order and we have presented some relaxation procedure to infer the interpretation of a term.

We now discuss some possible future works.

- It would be of interest to develop tractable (decidable) techniques to characterize the complexity classes BFF_i, at least for $i = 2$. Due to the intractability of the synthesis mentioned above, we have no expectation to solve this problem using interpretation methods. However other implicit complexity techniques such as tiering or light logics are candidates for solving this issue.
- Space issues were not discussed in this paper as there is no complexity theory for higher order polynomial space. However one could be interested in certifying program space properties. In

analogy with the usual first order theory, a suitable option could be to consider (possibly non-terminating) terms admitting a non strict polynomial interpretation. By non strict, we mean, for example, that the last rule of Definition 3.5 can be replaced by:

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

This would correspond to the notion of quasi-interpretation on first order programs [BMM11]. Termination is lost as the term $\texttt{letRec f} = \texttt{f}$ could be interpreted by $1 \oplus \Lambda F.F$. However a result equivalent to Lemma 3.9 holds: we still keep an upper bound on the interpretation of any derived term (hence on the "size" of such a term).

REFERENCES

[BL12] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. In *Computer Science Logic (CSL'12)*, volume 16 of *LIPIcs*, pages 62–76, 2012.

[BL16] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. Inf. Comput., volume 248, pages 56–81, 2016.

[BMM11] Guillaume Bonfante, Jean-Yves Marion and Jean-Yves Moyen. Quasi-interpretations a way to control resources. In *Theoretical Computer Science*, volume 412, pages 2776–2796, Elsevier, 2011.

[CL92] Adam Cichon and Pierre Lescanne. Polynomial interpretations and the complexity of algorithms. In *Journal of Automated Reasoning*, volume 34(4), pages 325–363, 2005.

[CMPU05] Evelyne Contejean, Claude Marché, Ana Paula Tomás and Xavier Urbain. Higher-order interpretations and program complexity. *International Conference on Automated Deduction*, Springer, pages 139–147, 1992.

[CK89] 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.

[Con73] Robert L. Constable. Type two computational complexity. In *Proc. 5th annual ACM Symposium on Theory of Computing*, pages 108–121. ACM, 1973.

[Fér14] Hugo Férée. *Complexité d'ordre supérieur et analyse récursive. (Higher order complexity and computable analysis)*. PhD thesis, University of Lorraine, Nancy, France, 2014.

[FHHP15] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations In *Theor. Comput. Sci.*, volume 585, pages 41–54, 2015.

[HP17] Emmanuel Hainry and Romain Péchoux. Higher order interpretation for higher order complexity In *21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 269–285, 2017.

[IKR02] Robert J. Irwin, Bruce M. Kapron, and James S. Royer. On characterizations of the basic feasible functionals (part II). Technical report, Syracuse University, 2002.

[IRK01] 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.

[KC10] 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.

[Ko91] Ker-I Ko. *Complexity Theory of Real Functions*. Birkhäuser, 1991.

[KS01] William A Kirk and Brailey Sims. *Handbook of metric fixed point theory*. Springer, 2001.

[Lan79] D.S. Lankford. On proving term rewriting systems are noetherian. Technical report, 1979.

[Meh74] Kurt Mehlhorn. Polynomial and abstract subrecursive classes. In *Proc. 6th annual ACM Symposium on Theory of Computing*, pages 96–109. ACM, 1974.

[Mit96] John C. Mitchell. *Foundations for programming languages*. MIT press Cambridge, 1996.

[MN70] Zohar Manna and Steven Ness. On the termination of Markov algorithms. In *Third Hawaii International Conference on System Science*, pages 789–792, 1970.

[Péc13] Romain Péchoux. Synthesis of sup-interpretations: A survey. *Theor. Comput. Sci.*, 467:30–52, 2013.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[VdP93] 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.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.