# Efficient first order functional program interpreter with time bound certifications

J.-Y. Marion J.-Y. Moyen *

June 6, 2000

**Abstract**

We demonstrate that the class of first order functional programs over lists which terminate by multiset path ordering and admit a polynomial quasi-interpretation, is exactly the class of function computable in polynomial time. The interest of this result lies (i) on the simplicity of the conditions on programs to certify their complexity, (ii) on the fact that an important class of natural programs is captured, (iii) and on potential applications on program optimizations.

## 1 Introduction

This paper is part of a general investigation on the implicit complexity of a specification. To illustrate what we mean, we write below the recursive rules that computes the longest common subsequences of two words. More precisely, given two strings $u = u_1 \cdots u_m$ and $v = v_1 \cdots v_n$ of $\{0,1\}^*$, a common subsequence of length $k$ is defined by two sequences of indices $i_1 < \cdots < i_k$ and $j_1 < \cdots < j_k$ satisfying $u_{i_q} = v_{j_q}$.

$$\begin{aligned}
\mathtt{lcs}(\epsilon, y) &\to \mathbf{0} \\
\mathtt{lcs}(x, \epsilon) &\to \mathbf{0} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y)) &\to \mathtt{lcs}(x, y) + 1 \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathtt{max}(\mathtt{lcs}(x, \mathbf{j}(y)), \mathtt{lcs}(\mathbf{i}(x), y)) \qquad \mathbf{i} \neq \mathbf{j}
\end{aligned}$$

The number of recursive calls is exponential because of the recomputing values. So, the execution of this specification would be unrealistic, and moreover, it is well-known that this problem is efficiently solved by a dynamic programming algorithm. Our purpose is to analyse a source program, as the specification above, to determine the computational complexity of the function computed. The next step is to generate an efficient implementation of the function denoted by the source program. Practically, we address the question of the efficiency of a program which is

*Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France, {marionjy,moyen}@loria.fr

a crucial, in particular in the context of semi-automatic program constructions. For example, Benzinger [4] has developed a prototype which analyses the complexity of a program extracted from a Nuprl proof. The same problematic motivates Crary and Weirich in [12] which introduce a type system to take into account the resource involved in a computation, see also [18]. Our approach is different because the complexity analysis must diagnose the intentional behaviour of data, and thereby help us to interpret programs more efficiently.

In this paper, we present an interpreter which makes online memoizing and runs in polynomial time on a non-trivial class of first-order functional programs over lists. The class of programs involved consists of programs which (i) terminate by *multiset path ordering* and (ii) admit a polynomial quasi-interpretation.

This interpreter is similar to the interpreter of cons-free WHILE programs defined by Jones [21, 22, 23]. This technique was initiated by Cook in [11] to demonstrate that languages which are recognised by 2-way pushdown automata are exactly polynomial time languages.

The program analysis combines and depends on one hand *the predicative analysis of recursion* and on the other on *term orderings*.

The predicative analysis of recursion was initiated by Bellantoni and Cook [2] and by Leivant [27, 26] which caracterized polynomial time computable functions (PTIME). The analysis permits us to separate off the active data of a recursion from the others, by assigning tier to data. In essence, those characterisations are extensional. So a lot of polynomial time algorithms are ruled out by the mechanism of data tiering. These intentional gaps were first coped by Caseiro in [6] and then by K. Aehlig and H. Schwichtenberg [1], Bellantoni, Niggl and Schwichtenberg [3], Hofmann [17, 19], and Leivant [28] in their studies on characterizations of PTIME by mean of higher type recursions.

Term orderings play a central role in proving the termination of term rewriting systems, and of programs that we shall consider. We refer to the comprehensive survey [14] of Dershowitz. *Multiset path ordering* (MPO) was introduced by Dershowitz [13]. A predicative analysis of recursion leads in [29] to the introduction of the term ordering *light multiset path ordering* ($LMPO$). It was established that $LMPO$ captures a significative class of algorithms. However, because of the underlying data-tiering, it was not permitted to iterate non-size increasing functions [17], for example.

The solution proposed in this paper is the outgrowth of the work [29] on $LMPO$[1] and includes several improvements. First, the new approach does not refer to data-tiering but uses it to build a predicative analysis which leads to a time resource bound certification. Second, the definition of the class of programs is much simpler and is closer to programming practice than the definition of $LMPO$. Third, the class of programs delineated allows iteration on non-size increasing functions and recursive template like `lcs` above.

[1] We presuppose no familiarity with [29]

# 2 First order functional programming

We shall consider first order programs over constructors which resemble to ELAN programs, or to a first-order restriction of HASKELL or ML programs. Throughout the following discussion, we consider four disjoint sets $\mathcal{X}, \mathcal{F}, \mathcal{C}, \mathcal{S}$ of variables, function symbols, constructors and sorts.

## 2.1 Syntax of programs

**Definition 1.** The sets of terms, patterns and function rules are defined in the following way:

| | | | |
|---|---|---|---|
| `(Constructor terms)` | $\mathcal{T}(\mathcal{C}) \ni u$ | $::=$ | $\mathbf{c}(u_1, \cdots, u_n)$ |
| `(Ground terms)` | $\mathcal{T}(\mathcal{C}, \mathcal{F}) \ni s$ | $::=$ | $\mathbf{c}(s_1, \cdots, s_n) \mid \mathtt{f}(s_1, \cdots, s_n)$ |
| `(terms)` | $\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t$ | $::=$ | $x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n)$ |
| `(patterns)` | $\mathcal{P} \ni p$ | $::=$ | $\mathbf{c}(p_1, \cdots, p_n) \mid x$ |
| `(rules)` | $\mathcal{D} \ni d$ | $::=$ | $\mathtt{f}(p_1, \cdots, p_n) \rightarrow t$ |

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$[2].

**Definition 2.** An untyped program is a quadruplet $main = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ such that:

- $\mathcal{E}$ is a set of $\mathcal{D}$-rules.
- Each variable in the right-hand side of a rule also appears in the left hand side of the same rule.
- `main` is the main function symbol of *main*.

The size $|t|$ of a term $t$ is the number of symbols in $t$. It is defined by $|c| = 1$ and $|f(t_1, \cdots, t_n)| = 1 + \sum_{i=1}^{n} |t_i|$.

## 2.2 Typing programs

Throughout, we consider typed programs over multi-sorted algebras.

**Definition 3.** Let $\mathcal{S} = \{\mathtt{s}_1, \cdots, \mathtt{s}_n\}$ be a set of sorts. The set of types builds upon $\mathcal{S}$ is $\mathit{Types} \ni \tau ::= \mathtt{s} \rightarrow \tau | \mathtt{s}$

Notice that each type is of the form $\mathtt{s}_1 \rightarrow (\cdots((\mathtt{s}_n) \rightarrow \mathtt{s})\cdots)$ and so denotes functions. We write it $(\mathtt{s}_1, \cdots, \mathtt{s}_n) \rightarrow \mathtt{s}$.

**Definition 4.** A typing environment is a mapping $\Gamma$ from $\mathcal{X} \bigcup \mathcal{C} \bigcup \mathcal{F}$ to *Types* which assigns to each symbol $f$ of arity $n > 0$ a type $\Gamma(f) = (\mathtt{s}_1, \cdots, \mathtt{s}_n) \rightarrow \mathtt{s}$, and, to each symbol of arity 0 a sort.

**Definition 5.** Let $\Gamma$ be a typing environment. A term $t \in \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X})$ is of type of $\mathtt{s}$, noted $t : \mathtt{s}$, iff:

- $t$ is a symbol of arity 0 and $\Gamma(t) = \mathtt{s}$.
- $t = f(t_1, \cdots, t_n)$, $t_i : \mathtt{s}_i$ and $\Gamma(f) = (\mathtt{s}_1, \cdots, \mathtt{s}_n) \rightarrow \mathtt{s}$.

**Definition 6.** A typed program is a sextuplet $main = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ such that $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ is an untyped program and for each rule $\mathtt{f}(p_1, \cdots, p_n) \rightarrow t$, the type of $\mathtt{f}(p_1, \cdots, p_n)$ and of $t$ are the same with respect to $\Gamma$.

[2] We shall use type writer font for function symbol and bold face font for constructors.

All along the paper, we assume that we are talking about a program *main*. We shall not explicitly mention $main = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$. also, $f : \Gamma(f)$ is an abbreviation for $f$ is of type $\Gamma(f)$.

**Example 7.** Given a list *list* of type $\mathtt{List(Nat)}$, $\mathtt{sort}(\mathit{list})$ sorts the element of *list*. The algorithm is the insertion sort. The sorts are $\mathcal{S} = \{\mathtt{bool}, \mathtt{Nat}, \mathtt{List}(\alpha)\}$ where $\alpha$ is a type variable, *i.e.* can be instantiated to any type $\mathtt{s} \in \mathcal{S}$. (This construction does not belong to our programming language but is admissible.) Constructors are $\mathcal{C} = \{\mathbf{tt} : \mathtt{bool}, \mathbf{ff} : \mathtt{bool}, \mathbf{0} : \mathtt{Nat}, \mathbf{suc} : \mathtt{Nat} \to \mathtt{Nat}, \mathbf{nil} : \mathtt{List}(\alpha), \mathbf{cons} : \alpha \times \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)\}$.

$\mathtt{if\_then\_else} : \mathtt{bool}, \alpha, \alpha \to \alpha$

$$\mathtt{if}\ \mathbf{tt}\ \mathtt{then}\ x\ \mathtt{else}\ y \to x$$
$$\mathtt{if}\ \mathbf{ff}\ \mathtt{then}\ x\ \mathtt{else}\ y \to y$$

$\_\mathtt{<}\_ : \mathtt{Nat}, \mathtt{Nat} \to \mathtt{bool}$

$$\begin{aligned} \mathbf{0} < \mathbf{suc}(y) &\to \mathbf{tt} \\ x < \mathbf{0} &\to \mathbf{ff} \\ \mathbf{suc}(x) < \mathbf{suc}(y) &\to x < y \end{aligned}$$

$\mathtt{insert} : \mathtt{Nat}, \mathtt{List(Nat)} \to \mathtt{List(Nat)}$

$$\begin{aligned} \mathtt{insert}(a, \mathbf{nil}) &\to \mathbf{cons}(a, \mathbf{nil}) \\ \mathtt{insert}(a, \mathbf{cons}(b, l)) &\to \mathtt{if}\ a < b\ \mathtt{then}\ \mathbf{cons}(a, \mathbf{cons}(b, l)) \\ &\qquad \mathtt{else}\ \mathbf{cons}(b, \mathtt{insert}(a, l)) \end{aligned}$$

$\mathtt{sort} : \mathtt{List(Nat)} \to \mathtt{List(Nat)}$

$$\begin{aligned} \mathtt{sort}(\mathbf{nil}) &\to \mathbf{nil} \\ \mathtt{sort}(\mathbf{cons}(a, l)) &\to \mathtt{insert}(a, \mathtt{sort}(l)) \end{aligned}$$

## 2.3 Semantic

The set $\mathcal{E}$ of rules induces a rewrite system. We recall briefly some basic notions of rewriting theories. For further details, one might consult Dershowitz and Jouannaud's survey [15]. The rewriting relation $\to$ induced by a program *main* is defined as follows $t \to s$ if $s$ is obtained from $t$ by applying one of the rules of $\mathcal{E}$. The relation $\overset{+}{\to}$ ($\overset{*}{\to}$) is the transitive (reflexive-transitive) closure of $\to$. Lastly, $t \overset{!}{\to} s$ means that $t \overset{*}{\to} s$ and $s$ is in normal form, *i.e.* no other rule may be applied. A ground (constructor) substitution is a substitution from $\mathcal{X}$ to $\mathcal{T}(\mathcal{C}, \mathcal{F})$ (resp. $\mathcal{T}(\mathcal{C})$).

Since our intention is to interpret a program by a function, an important property is that each term has a unique normal form, if it exists. Henceforth, we just consider *confluent* programs, that is programs for which the induced relation $\to$ is confluent. In particular, our definition embeds orthogonal programs which are confluent [20]. A program is orthogonal if in addition there is no overlapping rules.

We now give the semantic of (confluent) programs, based on the "standard" interpretation. The domain of interpretation is the constructor algebra $\mathcal{T}(\mathcal{C})$. Hence, we only consider normal forms in $\mathcal{T}(\mathcal{C})$ as being defined. Closed terms are interpreted by elements of the algebra $\mathcal{T}(\mathcal{C})/\mathcal{E}$, that is the initial algebra of the class of models of the set of program rules $\mathcal{E}$.

**Definition 8.** The semantic of a type $\mathtt{s}$ is the set of constructor terms whose type is $\mathtt{s}$. That is $[\![\mathtt{s}]\!] = \{t : \mathtt{s} \mid t \in \mathcal{T}(\mathcal{C})\}$.

**Definition 9.** Let *main* be a program and $\Gamma(\mathtt{main}) = (\mathtt{s}_1, \cdots, \mathtt{s}_n) \to \mathtt{s}$. The function computed by *main* is $[\![\mathtt{main}]\!] : [\![\mathtt{s}_1]\!] \times \cdots \times [\![\mathtt{s}_n]\!] \mapsto [\![\mathtt{s}]\!]$ which is defined as follows. For all $u_i \in [\![\mathtt{s}_i]\!], [\![\mathtt{main}]\!](u_1, \cdots, u_n) = v$ iff $\mathtt{main}(u_1, \cdots, u_n) \stackrel{!}{\to} v$ and $v \in [\![\mathtt{s}]\!]$, otherwise $[\![\mathtt{main}]\!](u_1, \cdots, u_n)$ is undefined.

# 3 Termination orderings

## 3.1 Multiset Path Ordering

*Multiset Path Orderings* (MPO) is a syntactic termination ordering which was introduced by Plaisted [31] and Dershowitz [13]. MPO is widely employed to prove program terminations. We briefly describe it together with some basic notions which we shall use later on.

A multiset $M$ of terms of $\mathcal{T}(\mathcal{C}, \mathcal{F})$ is a finite mapping $M : \mathcal{T}(\mathcal{C}, \mathcal{F}) \mapsto \mathbf{N}$ which associates to each term $t$ the number $M(t)$ of terms $t$ in $M$. Suppose that $\mathcal{T}(\mathcal{C}, \mathcal{F})$ is ordered by $\prec$. This ordering $\prec$ induces an ordering $\prec^m$ on term multisets.

**Definition 10.** $M \prec^m N$ iff $M \neq N$ and if there is $s \in \mathcal{T}(\mathcal{C}, \mathcal{F})$ such that $M(s) > N(s)$ then there is $t \in \mathcal{T}(\mathcal{C}, \mathcal{F})$ such that $s \prec t$ and $M(t) < N(t)$.

A precedence $\preceq_{\mathcal{F}}$ (strict precedence $\prec_{\mathcal{F}}$) is quasi-ordering (ordering) on the set $\mathcal{F}$ of function symbols. It is worth noticing that we shall consider constructors as minimal symbols with respect to $\preceq_{\mathcal{F}}$.

**Definition 11.** Let $\preceq_{\mathcal{F}}$ be a precedence on $\mathcal{F}$. The multiset path ordering $\prec_{mpo}$ is defined recursively by

1. $s \prec_{mpo} \mathbf{c}(\cdots, t_i, \cdots)$ if $\mathbf{c} \in \mathcal{C}$ and $s \preceq_{mpo} t_i$
2. $s \prec_{mpo} f(\cdots, t_i, \cdots)$, if $\mathtt{f} \in \mathcal{F}$ and $s \preceq_{mpo} t_i$
3. $\mathbf{c}(s_1, \cdots, s_n) \prec_{mpo} \mathtt{f}(t_1, \cdots, t_m)$ if $s_i \prec_{mpo} \mathtt{f}(t_1, \cdots, t_m)$, for $i \leq n$, and $\mathbf{c} \in \mathcal{C}$, $\mathtt{f} \in \mathcal{F}$. Note that $\mathbf{c}$ can be a 0-ary.
4. $\mathtt{g}(s_1, \cdots, s_m) \prec_{mpo} \mathtt{f}(t_1, \cdots, t_n)$, if $s_i \prec_{mpo} \mathtt{f}(t_1, \cdots, t_n)$, for $i \leq m$, and $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$.
5. $\mathtt{g}(s_1, \cdots, s_m) \prec_{mpo} \mathtt{f}(t_1, \cdots, t_n)$ if $\{s_1, \cdots, s_m\} \prec^m_{mpo} \{t_1, \cdots, t_n\}$ and $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$.

where $\preceq_{mpo} = \prec_{mpo} \bigcup \approx$.

### 3.2 A restriction : $MPO'$

We introduce a termination ordering $MPO'$ by restricting the multiset ordering of $MPO$. We shall see that both orderings define the class of primitive recursive functions.

**Definition 12.** Let $M = \{m_1, \cdots, m_p\}$ and $N = \{n_1, \cdots, n_p\}$ be two multisets with the same number of elements, $M \prec^{m'} N$ iff $M \neq N$ and there exists a permutation $\pi$ such that:

- $\exists i \leq p$ such that $m_i < n_{\pi(i)}$
- $\forall j \leq p, m_j \leq n_{\pi(j)}$

**Definition 13.** The ordering MPO' is defined by replacing $\prec^m$ by $\prec^{m'}$ in definition 11.

**Proposition 14.** *$\prec_{mpo}$ is an extension of $\prec_{mpo'}$, that is $s \prec_{mpo'} t$ implies $s \prec_{mpo} t$.*

*Proof.* Immediate by observing that $\prec^m$ is an extension of $\prec^{m'}$. □

A program is terminating by $MPO'$ ($MPO$) if there is a precedence on $\mathcal{F}$ such that for each ground substitution $\sigma$ and each rule $l \to r$ of $\mathcal{E}$, we have $r\sigma \prec_{mpo'} l\sigma$ (resp. $\prec_{mpo}$).

**Examples 15.**

1. By putting `if_then_else` $\prec_{\mathcal{F}}$ `_<_` $\prec_{\mathcal{F}}$ `insert` $\prec_{\mathcal{F}}$ `sort`, we check that the program written in Example 7 terminates by $MPO'$.
2. The following program computes the exponential. It is convenient to identify $\mathbf{0}$ with 0 and $\mathbf{suc}(n)$ with $n+1$. Henceforth, we write $\mathbf{n}$ instead of $\mathbf{suc}^n(\mathbf{0})$. In the same way, $\mathbf{n}+\mathbf{m}$ stands for $\mathbf{suc}^{n+m}(\mathbf{0})$. The program is ordered by MPO' with $\mathtt{d} \prec_{\mathcal{F}} \mathtt{exp}$.

   $\mathtt{d} : \mathtt{Nat} \to \mathtt{Nat}$ and $[\![\mathtt{d}]\!](\mathbf{n}) = \mathbf{n}+\mathbf{n}$

   $$\mathtt{d}(\mathbf{0}) \to \mathbf{0}$$
   $$\mathtt{d}(\mathbf{suc}(x)) \to \mathbf{suc}(\mathbf{suc}(\mathtt{d}(x)))$$

   exp : $\mathtt{Nat} \to \mathtt{Nat}$ and $[\![\mathtt{exp}]\!](\mathbf{n}) = \mathbf{2^n}$

   $$\mathtt{exp}(\mathbf{0}) \to \mathbf{suc}(\mathbf{0})$$
   $$\mathtt{exp}(\mathbf{suc}(x)) \to \mathtt{d}(\mathtt{exp}(x))$$

Define Prog($MPO'$) as the set of programs which are terminated by $MPO'$. A program of Prog($MPO'$) is constructed following some templates which are defined by the ordering $MPO'$. It is a particular case of "un-nested multiple recursion". Typical examples of such templates are illustrated by `inf` in Example 7 and by `lcs` in Example 23(3).

Now, define $SPR$ as the following class of programs. A program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ is in $SPR$ if (i) there is a strict precedence $\prec_{\mathcal{F}}$ on the set $\mathcal{F}$ of defined function symbols, and (ii) each rule is of the following form:

Explicit definition assuming that $\mathtt{h} \in \mathcal{F}$ and $\mathtt{h}$ is in $t$, then $\mathtt{h} \prec_{\mathcal{F}} \mathtt{f}$

$$\mathtt{f}(p_1, \cdots, p_n) \to t$$

Primitive recursion assuming that $\mathtt{g}, \mathtt{h} \prec_{\mathcal{F}} \mathtt{f}$

$$\mathtt{f}(\mathbf{0}, x_1, \cdots, x_{n-1}) \to \mathtt{g}(x_1, \cdots, x_{n-1})$$
$$\mathtt{f}(\mathbf{suc}(t), x_1, \cdots, x_{n-1}) \to \mathtt{h}(t, x_1, \cdots, x_{n-1}, \mathtt{f}(t, x_1, \cdots, x_{n-1}))$$

It is not difficult to see that the class of functions $\mathbb{F}(SPR)$ computed by $SPR$ is exactly the class of primitive recursive functions.

On the other hand, it is routine to check that each $SPR$ rule is terminated by$MPO'$. Next, by proposition 14, each program which terminates by $MPO'$ also terminates by $MPO$. Hofbauer in [16] demonstrated that functions which terminate by $MPO$ are exactly primitive recursive functions. So as a corollary, we state

**Corollary 16.** *The class of functions $\mathbb{F}(MPO')$ computed by a confluent program which terminates by $MPO'$ is exactly the class of the primitive recursive functions.*

So, we have two sets of programs, Prog$(MPO')$ and Prog$(SPR)$, which are extensionally equivalent, *i.e.* $\mathbb{F}(MPO') = \mathbb{F}(()SPR)$. But, there are more programs in Prog$(MPO')$ than in Prog$(()SPR)$. More importantly, Colson showed, see [10], that there is no program in $SPR$ which computes the minimum of two integers $n$ and $m$ in $\inf(n, m)$ steps. However, `_<_` defined in Example 7 provides such algorithm. So, Prog$(MPO')$ contains more "good" programs than $SPR$ because more algorithmic patterns are allowed. This remark raises the question of "intentional completeness"with respect to the class primitive recursion functions. This problems was coped by Péter in her monograph [30] by showing that a lot of recursion schema stay in the realm of primitive recursion. Simmons [33] studied what could be the rational beyond such results. A new approach about those questions, is suggested by Cichon and Weiermann in [7], and which contains also other references.

To summarise the discussion above, termination orderings, like $MPO'$, are a starting point to study intentional property of class of programs and the resource bounds to compute a function denoted by such a program.

# 4 Ordering and feasible computation

## 4.1 Data tiering

Bellantoni and Cook [2] and Leivant [27] characterized polynomial-time functions in syntactic way, unlike characterizations à la Cobham [9] and Ritchie [32]. (More information may be found in Clote's survey [8].)

Their approaches are based on the predicative analysis of recursion. The rational for this analysis is intuitively explained from the Example 15(2). The existence of the function $E$ denoted by `exp` is demonstrated by an induction on natural numbers. At some point of the proof, one have to assume that $E(n)$ is defined, in order to show the existence of $E(n+1)$.

That is, one have to assume that the term $\mathtt{exp(n)}$ can be reduced to $2^{\mathtt{n}}$. This reasoning is legitimate as long as one do not care about computations and resource bounds. Predicative analysis of recursion points out this hypothesis as the source of trouble in studying feasible computations. For this reason, a data ramification, or a data tiering, is introduced to control recursion by assigning to the result of a recursion a tier which is strictly lower than the tier of the recurrence parameter. Thus, it is not possible to use the result of a recursion as a recursion parameter because of the tier difference. And so, primitive recursion is tame.

## 4.2 Light Multiset Path Ordering

The term ordering Light multiset path ordering (LMPO) was introduced in [29] to encompass a broad class of algorithms delineating polynomial time functions. $LMPO$ is a restriction of $MPO'$ based on the predicative analysis of recursion. For this, each argument of a function symbol has a valency which predicts their intentional uses.

**Definition 17.** A valency of a function symbol $\mathtt{f}$ of arity $n$ is a mapping $\nu(\mathtt{f}) : \{1, \cdots, n\} \mapsto \{0, 1\}$.

The valency of a function symbol $\mathtt{f}$ indicates how to compare arguments of $\mathtt{f}$ with respect to $LMPO$. Technically, valencies are like the notion of status which was suggested by Kamin and Levy [24].

**Definition 18.** Let $\nu$ be a valency function on the set $\mathcal{F}$ of function symbols. A permutation $\pi$ over $\{1, \cdots, n\}$ respects the valency of $\mathtt{f}$ and $\mathtt{g}$ if the arity of $\mathtt{f}$ and $\mathtt{g}$ is $n$ and $\nu(\mathtt{g}, i) = \nu(\mathtt{f}, \pi(i))$, for all $i \leq n$.

**Definition 19.** Let $\prec_0$ and $\prec_1$ be two term orderings. These orderings are lifted to an ordering over $n$-uplets of terms which respects the valency function $\nu$ on $\mathcal{F}$ as follows. $\{s_1, \cdots, s_n\} \prec^{\nu}_{\mathtt{g},\mathtt{f}} \{t_1, \cdots, t_n\}$ iff there is a permutation $\pi$ which respects the valency of $\mathtt{f}$ and $\mathtt{g}$ and which satisfies that there is $j \leq n$ such that $\nu(\mathtt{g}, j) = 1$ and $s_j \prec_1 t_{\pi(j)}$, and for all $i \leq n$, $s_i \preceq_{\nu(\mathtt{g},i)} t_{\pi(i)}$.

The above ordering on $n$-uplets of terms is a restriction of the multiset ordering, as defined in 10, induced by the transitive closure of $\prec_1 \bigcup \prec_0$.

**Definition 20.** Let $\approx_{\mathcal{F}}$ be a equivalence relation on $\mathcal{F}$. A *permutative congruence* $\approx$ which respects the valency $\nu$ is the smallest homeomorphic extension of $\approx_{\mathcal{F}}$ to terms of $\mathcal{T}(\mathcal{C}, \mathcal{F})$ which satisfies:

1. $\mathbf{c}(s_1, \cdots, s_n) \approx \mathbf{c}(t_1, \cdots, t_n)$ if $\mathbf{c} \in \mathcal{C}$ and $s_i \approx t_i$ for all $i \leq n$.
2. $\mathtt{f}(t_1, \cdots, t_n) \approx \mathtt{g}(s_1, \cdots, s_n)$ if $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, $t_i \approx s_{\pi(i)}$ for some permutation $\pi$ which respects the valency of $\mathtt{f}$ and $\mathtt{g}$.

**Definition 21.** Let $\preceq_{\mathcal{F}}$ be a precedence and $\nu$ be a valency function on $\mathcal{F}$. The *ligth multiset path ordering* is a pair $(\prec_k)_{k=0,1}$ of orderings which is recursively defined on $\mathcal{T}(\mathcal{C}, \mathcal{F})$ as follows:

1. $s \prec_k \mathbf{c}(\cdots, t_i, \cdots)$ if $s \preceq_k t_i$ and $\mathbf{c} \in \mathcal{C}$.
2. $s \prec_k \mathtt{f}(\cdots, t_i, \cdots)$ if $s \preceq_{\nu(\mathtt{f},i)} t_i$ and $k \leq \nu(\mathtt{f}, i)$.
3. $\mathbf{c}(s_1, \cdots, s_n) \prec_k \mathtt{f}(t_1, \cdots, t_m)$ if $\mathbf{c} \in \mathcal{C}$, $\mathtt{f} \in \mathcal{F}$ and $s_i \prec_k \mathtt{f}(t_1, \cdots, t_m)$, for each $i \leq n$. Note that $\mathbf{c}$ can be a 0-ary.

4. $\mathtt{g}(s_1, \cdots, s_n) \prec_k \mathtt{f}(t_1, \cdots, t_m)$ if $s_i \prec_{\max(k,\nu(\mathtt{g},i))} \mathtt{f}(t_1, \cdots, t_m)$, for each $i \leq n$ and if $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$.
5. $\mathtt{g}(s_1, \cdots, s_n) \prec_0 \mathtt{f}(t_1, \cdots, t_n)$ if $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ and $\{s_1, \cdots, s_n\} \prec^{\nu}_{\mathtt{g},\mathtt{f}} \{t_1, \cdots, t_n\}$

where $\preceq_k = \prec_k \bigcup \approx$

**Definition 22.** A program *main* is terminating by *LMPO* if there is a valency function $\nu$ such that for each rule $(l \to r) \in \mathcal{E}$, and each constructor substitution $\sigma$, we have $r\sigma \prec_0 l\sigma$. A *LMPO-program* is a program which terminates by *LMPO*.

**Examples 23.** Throughout, we use the following convention to indicate the valency of a function argument. The arguments of valency 1 are the first arguments of $\mathtt{f}$ and the last ones are of valency 0, and are separated by ';', like normal/safe arguments of [2].

1. The following program is terminating using either MPO, MPO' or LMPO.

   $\mathtt{add} : \mathtt{Nat}_1, \mathtt{Nat}_0 \to \mathtt{Nat}$ and $[\![\mathtt{add}]\!](\mathbf{x}, \mathbf{y}) = \mathbf{x} + \mathbf{y}$

$$\begin{aligned} \mathtt{add}(\mathbf{0}; y) &\to y \\ \mathtt{add}(\mathbf{suc}(x); y) &\to \mathbf{suc}(\mathtt{add}(x; y)) \end{aligned}$$

   $\mathtt{mult} : \mathtt{Nat}_1, \mathtt{Nat}_1 \to \mathtt{Nat}$ and $[\![\mathtt{mult}]\!](\mathbf{x}, \mathbf{y}) = \mathbf{x} \times \mathbf{y}$

$$\begin{aligned} \mathtt{mult}(\mathbf{0}, y;) &\to \mathbf{0} \\ \mathtt{mult}(\mathbf{suc}(x), y;) &\to \mathtt{add}(y; \mathtt{mult}(x, y;)) \end{aligned}$$

2. The exponential program given in Example 15(2) is terminating using either MPO or MPO' but not with LMPO. Indeed, the second equation will force $\nu(\mathtt{d}, 1) = 1$ and so the fourth equation cannot be ordered because $\mathtt{exp}(x) \not\preceq_1 \mathtt{exp}(\mathbf{suc}(x))$.
3. Take again the problem of computing the longest common subsequence of two words. To write a *LMPO*-program to solve this problem, we encode binary words by the sort $\mathtt{word}$ generated by the constructors $\{\epsilon : \mathtt{word}, \mathbf{a} : \mathtt{word} \to \mathtt{word}, \mathbf{b} : \mathtt{word} \to \mathtt{word}\}$. Then, we write the recursive solution of the problem.

   $\mathtt{max} : \mathtt{word}_1, \mathtt{Nat}_0, \mathtt{Nat}_0 \to \mathtt{Nat}$

$$\begin{aligned} \mathtt{max}(x; n, \mathbf{0}) &\to n \\ \mathtt{max}(x; \mathbf{0}, m) &\to m \\ \mathtt{max}(\mathbf{i}(x); \mathbf{suc}(n), \mathbf{suc}(m)) &\to \mathbf{suc}(\mathtt{max}(x; n, m)) \qquad \mathbf{i} \in \{\mathbf{a}, \mathbf{b}\} \end{aligned}$$

   $\mathtt{lcs} : \mathtt{word}_1, \mathtt{word}_1 \to \mathtt{Nat}$

$$\begin{aligned} \mathtt{lcs}(x, \epsilon;) &\to \mathbf{0} \\ \mathtt{lcs}(\epsilon, y;) &\to \mathbf{0} \\ \mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y);) &\to \mathbf{suc}(\mathtt{lcs}(x, y;)) \\ \mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y);) &\to \mathtt{max}(\mathbf{i}(x); \mathtt{lcs}(x, \mathbf{j}(y);), \mathtt{lcs}(\mathbf{i}(x), y;)) \qquad \mathbf{i} \neq \mathbf{j} \end{aligned}$$

The program is ordered by $\prec_0$ by putting $\mathtt{max} \prec_{\mathcal{F}} \mathtt{lcs}$. It is worth noticing that it is necessary that max has a extra argument of valency 1. The definition is un-natural. Similar defect is observable with the insertion sort in Example 7. The reason was clearly analysed by Caseiro [6], Hofmamm [17] and K. Aehlig and H. Schwichtenberg [1]. Data-tiering prevents from iterating on a result (*i.e.* on the value $\mathtt{lcs}(x, \mathbf{j}(y);)$ of the example above) previously obtained by iteration, even if the function (*i.e.* max) does not increase the size of the output. We remedy to this defect in the next Section 5.

### 4.3 LMPO-interpreter

**Definition 24.** A sort $\mathtt{s} \in \mathcal{S}$ is simple if each constructor $\mathbf{c} \in \mathcal{C}$ of type $(\mathtt{s}_1, \cdots, \mathtt{s}_n) \to \mathtt{s}$, the sort $\mathtt{s}$ appears at most once in $(\mathtt{s}_1, \cdots, \mathtt{s}_n)$. A *LMPO*-program over a simple constructor signature is a *LMPO*-program in which each sort is simple.

The sorts bool,Nat, and $\mathtt{List}(\alpha)$ where $\alpha$ is simple, are simple. The sort of binary trees is not simple.

**Theorem 25.** *Each LMPO-program over a simple constructor signature is computable in polynomial time, and conversely each polynomial time function is computed by a LMPO-program over a simple constructor signature.*

*Sketch of proof.* The detailed proof can be found in [29]. The second implication is proved by programming each function of the class $B$ of Bellantoni and Cook [2], by merely identifying safe (normal) arguments with arguments of valency 1 (resp. 0).

Conversely, a *LMPO*-program is evaluated by call-by-value interpreter with a cache memory. Whenever the interpreter wants the value of a call $\mathtt{f}(t_1, \cdots, t_n)$, it looks first in the cache. If $\mathtt{f}(t_1, \cdots, t_n)$ is not in the cache, then it computes $\mathtt{f}(t_1, \cdots, t_n)$ and stores the result in cache, so that the next call to $\mathtt{f}(t_1, \cdots, t_n)$ will be in the cache. The algorithm of the interpreter is presented in Figure 4.3.

The time-bound of the interpreter is obtained by establishing that the number of recursive calls is bounded by a polynomial in the height of the inputs. Since inputs are of simple sorts, the size of an input is at most quadratic in the height. Consequently, the running time of the interpreter is bounded by a polynomial in the size of the inputs.

It is necessary to memorise intermediate values because the length of a term rewriting can be exponential. Indeed, the length of a derivation of lcs, in Example 23(3), is exponential. Consequently, the interpreter can transform a program into an exponentially faster one. □

## 5 Quasi-interpretation and MPO'

### 5.1 Polynomial quasi interpretation

**Definition 26.** Let $\mathtt{f} \in \mathcal{F} \bigcup \mathcal{C}$ be either a function symbol or a constructor of arity $n$. A quasi-interpretation of $\mathtt{f}$ is a mapping $(\!|\mathtt{f}|\!) : \mathbf{N}^n \to \mathbf{N}$

Given a set of equations $\mathcal{E}$ and a ground substitution $\sigma$, $t \xrightarrow{!} v$ using the cache $C$ and obtaining the cache $C'$ is written:

$$\mathcal{E}, \sigma \vdash \langle C, t \rangle \rightarrow \langle C', v \rangle$$

$$\mathcal{E}, \emptyset \vdash \langle \emptyset, \mathtt{f}(t_1, \cdots, t_n) \rangle \rightarrow \langle C, v \rangle \text{ means that } [\![\mathtt{f}]\!](t_1, \cdots, t_n) = v$$

$$\frac{\sigma(x) = v}{\mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \cdots, t_n) \rangle \rightarrow \langle C_n, \mathbf{c}(v_1, \cdots, v_n) \rangle}$$

$$\frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \quad (\mathtt{f}(v_1, \cdots, v_n), v) \in C_n}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \rightarrow \langle C_n, v \rangle}$$

$$\frac{\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \quad \mathtt{f}(\vec{p}) \rightarrow r \in \mathcal{E} \quad p_i \sigma' = v_i \quad \mathcal{E}, \sigma' \vdash \langle C_n, r \rangle \rightarrow \langle C, v \rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \rightarrow \langle C \cup (\mathtt{f}(v_1, \cdots, v_n), v), v \rangle}$$

Figure 1: Call by value $LMPO$-interpreter with a cache

which satisfies

1. $(\!|\mathtt{f}|\!)$ is (non-strictly) increasing with respect to each argument.
2. $(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) \geq X_i$, for each $1 \leq i \leq n$.
3. $(\!|\mathbf{c}|\!) > 0$ for each 0-ary symbol $\mathbf{c}$.

We extend a quasi interpretation $(\!|\bullet|\!)$ to terms as follows :

$$(\!|\mathtt{f}(t_1, \cdots, t_n)|\!) = (\!|\mathtt{f}|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!))$$

**Definition 27.** $(\!|\bullet|\!)$ is a quasi-interpretation of a program *main* if for each rule $l \rightarrow r \in \mathcal{E}$ and for each constructor substitution $\sigma$, $(\!|r\sigma|\!) \leq (\!|l\sigma|\!)$.

**Definition 28.** A program *main* admits a polynomial quasi interpretation $(\!|\bullet|\!)$, if $(\!|\bullet|\!)$ is bounded by a polynomial and each constructor $\mathbf{c} \in \mathcal{C}$ is interpreted by $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + a$ where $a > 0$.

Clearly, a quasi-interpretation is not sufficient to assure the termination of a program. For example, take the rule $\mathtt{f}(x) \rightarrow \mathtt{f}(x)$ which has a quasi-interpretation but does not terminate.

*Remark 29.* Termination proofs by polynomial interpretations were proposed by Lankford [25]. The termination is assured by requiring that $(\!|\bullet|\!)$ is strictly increasing and also that for each rule, $(\!|r\sigma|\!) < (\!|l\sigma|\!)$. Bonfante, Cichon, Marion and Touzet in [5] established that functions computed by programs admitting a polynomial interpretation termination proof, depends on constructor interpretations. In particular, when constructors are interpreted by $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + a$ (as we do), the functions computed by such systems are exactly the functions computed in polynomial-time.

Let us end with a Lemma that we shall use later on.

**Lemma 30.** *Let $t$ and $s$ be two constructor terms of $\mathcal{T}(\mathcal{C})$. If $t \preceq_k s$ then $(\!|t|\!) < (\!|s|\!)$, for $k = 0, 1$.*

*Proof.* By induction on the size of $s$. □

## 5.2 Main result

**Definition 31.** The class of functions $\mathcal{F}^{\mathrm{poly}}(MPO')$ is the class of functions computed by a program which terminates by $MPO'$ and admits a polynomial quasi-interpretation.

**Theorem 32.** *$\mathcal{F}^{poly}(MPO')$ is exactly the class of functions computed in polynomial-time.*

*Proof.* In [2], Bellantoni and Cook have demonstrated that the class of $B$ functions is exactly the class of polynomial time functions. It is easy to see that each function of the class $B$ is ordered by $MPO'$. Then, Lemma 4.1 of [2] provides a quasi-interpretation. It follows that each polynomial time function is in $\mathcal{F}^{\mathrm{poly}}(MPO')$.

Conversely, each program is executed by the $LMPO$-interpreter described in Figure 4.3. Let us give an informal account on how to establish that the runtime is polynomially bounded. We lift each source program *main* to a program $\rceil main \lceil$ which is ordered by $LMPO$. Intuitively, the lifting mapping $\rceil\bullet\lceil$ is akin a polynomial reduction from $MPO'$-programs to $LMPO$-programs. Each rule of the program $\rceil main \lceil$ has two new arguments. The first argument plays the role of a virtual clock. The second one is a copy of the initial setting of the clock. The semantic of the source program *main* is preserved only if the virtual clock is set to a value greater than the quasi-interpretation. We see that if the quasi-interpretation is bounded by a polynomial, then the runtime of the execution of *main* is also polynomial by Theorem 25. The details of the proof are in Section 6. □

**Examples 33.**

1. Consider again the problem which computes the longest subsequence between two words. The rules of the Introduction which defined `lcs` are ordered by $MPO'$ and possesses the following polynomial quasi-interpretation : $(\!|\mathbf{0}|\!) = (\!|\epsilon|\!) = 1$, $(\!|\mathbf{suc}|\!)(X) = (\!|\mathbf{a}|\!)(X) = (\!|\mathbf{b}|\!)(X) = X + 1$, $(\!|\mathtt{max}|\!)(X, Y) = \max(X, Y)$, $(\!|\mathtt{lcs}|\!)(X, Y) = \max(X, Y)$. So, this program is evaluated in polynomial time. Finally, notice also that `lcs`-rules have no polynomial interpretation.

2. The insertion sort in Example 7 terminates by $MPO'$ and admits the following polynomial quasi-interpretation : $(\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) = (\!|\mathbf{nil}|\!) = (\!|\mathbf{0}|\!) = 1$, $(\!|\mathbf{suc}|\!)(X) = (\!|\mathbf{cons}|\!)(X) = X+1$, $(\!|\mathtt{if\_then\_else}|\!)(X,Y,Z) = max\{X,Y,Z\}$, $(\!|\mathtt{\_<\_}|\!)(X,Y) = max\{X,Y\}$, $(\!|\mathtt{insert}|\!)(X,Y) = X + Y + 1$, $(\!|\mathtt{sort}|\!)(X) = X$. Notice that $LMPO$ is not able to order the rules of the insertion sort, because of the data-tiering.
3. The definition of the exponential in Example 15(2) has no polynomial quasi-interpretation, because $(\!|\mathtt{d}|\!)(X) = 2 \times X$ and, we obtain the inequality $(\!|\mathtt{exp}|\!)(X + c) \geq 2 \times (\!|\mathtt{exp}|\!)(X)$ whose solution is exponential.

# 6 Runtime analysis

Without loss of generality, we assume that each program contains the sort `Nat`.

## 6.1 Program lifting

Let $\preceq_{\mathcal{F}}$ be a precedence over $\mathcal{F}$. We define a lift mapping $\rceil\bullet\lceil$ which transforms each program $main = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ into a program $\rceil main\lceil = \langle \rceil\mathcal{X}\lceil, \mathcal{C}, \rceil\mathcal{F}\lceil, \rceil\mathcal{E}\lceil \rangle$ where

- $\rceil\mathcal{X}\lceil = \mathcal{X} \cup \{X, Y\}$ where $X$ and $Y$ are two variables which are not in $\mathcal{X}$,
- The set of function symbols $\rceil\mathcal{F}\lceil$ consists of a new symbol $\mathtt{f}'$ of arity $n+2$, for each symbol $\mathtt{f} \in \mathcal{F}$ of arity $n > 0$.
- The set of rules $\rceil\mathcal{E}\lceil$ contains a rule

$$\mathtt{f}'(\mathbf{suc}(X), Y; p_1, \cdots, p_n) \to \rceil s \lceil^{\mathtt{f}}_{X,Y}$$

  for each rule $\mathtt{f}(p_1, \cdots, p_n) \to s$ of $\mathcal{E}$ and $\rceil s\lceil^{\mathtt{f}}_{X,Y}$ is defined below.

The lift mapping $\rceil\bullet\lceil^{\mathtt{f}}_{X,Y}$ over terms is defined as follows:

- $\rceil x\lceil^{\mathtt{f}}_{X,Y} = x$ for all $x \in \mathcal{X}$.
- $\rceil \mathbf{c}(t_1, \cdots, t_m)\lceil^{\mathtt{f}}_{X,Y} = \mathbf{c}(\rceil t_1\lceil^{\mathtt{f}}_{X,Y}, \cdots, \rceil t_m\lceil^{\mathtt{f}}_{X,Y})$ if $\mathbf{c} \in \mathcal{C}$.
- $\rceil \mathtt{g}(t_1, \cdots, t_m)\lceil^{\mathtt{f}}_{X,Y} = \mathtt{g}'(Y, Y, \rceil t_1\lceil^{\mathtt{f}}_{X,Y}, \cdots, \rceil t_m\lceil^{\mathtt{f}}_{X,Y})$, if $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$
- $\rceil \mathtt{g}(t_1, \cdots, t_n)\lceil^{\mathtt{f}}_{X,Y} = \mathtt{g}'(X, Y, \rceil t_1\lceil^{\mathtt{f}}_{X,Y}, \cdots, \rceil t_n\lceil^{\mathtt{f}}_{X,Y})$, if $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$

**Proposition 34.** *Let main be a $MPO'$-program, $\rceil main\lceil$ is a $LMPO$-program.*

*Proof.* We define the valency function $\nu$ over $\rceil\mathcal{F}\lceil$ by $\nu(\mathtt{f}', 1) = \nu(\mathtt{f}', 2) = 1$ and $\nu(\mathtt{f}', 3+i) = 0$, $i < n$, for each symbol $\mathtt{f} \in \mathcal{F}$ of arity $n > 0$.

Now, consider a rule $t = \mathtt{f}(p_1, \cdots, p_n) \to s$ of $\mathcal{E}$. Suppose that $\sigma$ is a constructor substitution. We show that if $s\sigma \prec_{mpo'} t\sigma$ then $\rceil s\lceil^{\mathtt{f}}_{X,Y}\sigma' \prec_0 \mathtt{f}'(\mathbf{suc}(X), Y; p_1, \cdots, p_n)\sigma' = T$ where $\sigma'$ extends $\sigma$. The proof goes by induction on the size of the right hand side term $s$.

If $s\sigma = \mathbf{c}(u_1, \cdots, u_m)$, $\mathbf{c} \in \mathcal{C}$, then $\rceil s \lceil^{\mathtt{f}}_{X,Y} = \mathbf{c}(\rceil u_1 \lceil^{\mathtt{f}}_{X,Y}, \cdots, \rceil u_m \lceil^{\mathtt{f}}_{X,Y})$. By MPO' definition, $u_i\sigma \prec_{mpo'} t\sigma$ for each $1 \le i \le m$. So, by induction hypothesis, $\rceil u_i \lceil^{\mathtt{f}}_{X,Y}\sigma' \prec_0 T$. So, by definition of LMPO (rule 3), $\rceil s \lceil^{\mathtt{f}}_{X,Y}\sigma' \prec_0 T$.

If $s\sigma = \mathtt{g}(u_1, \cdots, u_m)$, where $\mathtt{g} \in \mathcal{F}$ and $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$, then $\rceil s \lceil^{\mathtt{f}}_{X,Y} = \mathtt{g}'(Y, Y; \rceil u_1 \lceil^{\mathtt{f}}_{X,Y}, \cdots, \rceil u_m \lceil^{\mathtt{f}}_{X,Y})$. By MPO' definition, $u_i\sigma \prec_{mpo'} t\sigma$ for each $1 \le i \le m$. So, by induction hypothesis, $\rceil u_i \lceil^{\mathtt{f}}_{X,Y}\sigma' \prec_0 T$. We have $\sigma'(Y) \prec_1 T$. It follows by definition of LMPO (rule 4) that $\rceil s \lceil^{\mathtt{f}}_{X,Y}\sigma' \prec_0 T$.

If $s\sigma = \mathtt{g}(u_1, \cdots, u_n)$, where $\mathtt{g} \in \mathcal{F}$ and $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$, then $\rceil s \lceil^{\mathtt{f}}_{X,Y} = \mathtt{g}'(X, Y; \rceil u_1 \lceil^{\mathtt{f}}_{X,Y}, \cdots, \rceil u_n \lceil^{\mathtt{f}}_{X,Y})$. By definition of MPO', there exists a permutation $\pi$ such that $u_i\sigma \preceq_{mpo'} p_{\pi(i)}\sigma$ for each $1 \le i \le n$. Since $u_i$'s and $p_i$'s are terms of $\mathcal{T}(\mathcal{T}(\mathcal{C}, \mathcal{X}))$, we have $\rceil u_i \lceil^{\mathtt{f}}_{X,Y}\sigma' \preceq_0 p_{\pi(i)}\sigma'$. Next, we have $\sigma'(X) \prec_1 \mathbf{suc}(\sigma'(X))$. The definition of LMPO (rule 5) gives: $\rceil s \lceil^{\mathtt{f}}_{X,Y}\sigma' \prec_0 T$.

Since each rule of *main* is ordered by $MPO'$, we have showed that each rule of $\rceil main \lceil$ is ordered by $LMPO$, which leads to the conclusion. □

## 6.2 Quasi-interpretations are virtual clocks

We define $\rfloor \bullet \lfloor$ as the inverse of the term lifting $\rceil \bullet \lceil^{\mathtt{f}}_{X,Y}$ as follows.

- $\rfloor x \lfloor = x$ if $x \in \mathcal{X}$.
- $\rfloor \mathbf{c}(t_1, \cdots, t_n) \lfloor = \mathbf{c}(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor)$ if $\mathbf{c} \in \mathcal{C}$.
- $\rfloor \mathtt{f}'(s, u; t_1, \cdots, t_n) \lfloor = \mathtt{f}(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor)$ if $\mathtt{f} \in \mathcal{F}$.

We see that $\rfloor\ \rceil t \lceil^{\mathtt{f}}_{X,Y} \lfloor = t$.

**Lemma 35.** *Let main $= \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a $MPO'$-program which admits a quasi-interpretation $(\!|\bullet|\!)$. Let $d$ be the greatest arity of a function symbol in main. Let $t$ be a ground term of $\mathcal{T}(\mathcal{C}, \rceil \mathcal{F} \lceil)$.*

*Suppose that for each subterm $f'(\mathbf{h_1}, \mathbf{h_2}; t_1, \cdots, t_n)$ of $t$, we have (i) $h_1 \ge \sum_{i=1}^{n} (\!|\rfloor t_i \lfloor|\!)$ and (ii) $h_2 \ge d \times (\!|f(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor)|\!)$.*

*If $\rfloor t \lfloor \to u$ then there exists a term $v$ such that*

1. *$\rfloor v \lfloor = u$ and $t \to v$.*
2. *For each subterm of $g'(\mathbf{h_1'}, \mathbf{h_2'}; v_1, \cdots, v_m)$ of $v$, we have (i) $h_1' \ge \sum_{i=1}^{m} (\!|\rfloor v_i \lfloor|\!)$ and (ii) $h_2' \ge d \times (\!|g(\rfloor v_1 \lfloor, \cdots, \rfloor v_m \lfloor)|\!)$.*

*Proof.* $\rfloor t \lfloor \to u$ means that there are a rule $e \equiv \mathtt{f}(p_1, \cdots, p_n) \to r$ in $\mathcal{E}$, a subterm $\mathtt{f}(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor)$ of $\rfloor t \lfloor$ and a constructor substitution $\sigma$ such that $\mathtt{f}(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor) = \mathtt{f}(p_1, \cdots, p_n)\sigma$ and $u = \rfloor t \lfloor [r\sigma / \mathtt{f}(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor)]$.

The rule $e$ is lifted to $\rceil e \lceil \equiv \mathtt{f}'(\mathbf{suc}(X), Y; p_1, \cdots, p_n) \to \rceil r \lceil^{\mathtt{f}}_{X,Y}$. There are $\mathbf{h_1}$, and $\mathbf{h_2}$ such that $\rfloor \mathtt{f}'(\mathbf{h_1}, \mathbf{h_2}; t_1, \cdots, t_n) \lfloor = \mathtt{f}(\rfloor t_1 \lfloor, \cdots, \rfloor t_n \lfloor)$. We have $h_1 > 0$ because quasi-interpretations are always strictly positive. It follows that there is a constructor substitution $\sigma'$ which extends $\sigma$ and such that $\sigma'(X) = \mathbf{h_1} - 1$ and $\sigma'(Y) = \mathbf{h_2}$. Therefore, we apply $\rceil e \lceil$ to $t$ and we obtain $v = t[\rceil r \lceil^{\mathtt{f}}_{X,Y}\sigma' / \mathtt{f}'(\mathbf{h_1}, \mathbf{h_2}; t_1, \cdots, t_n)]$. We get $\rfloor v \lfloor = u$.

It remains to establish (2). Take a subterm $\mathtt{g}'(\mathbf{h_1'}, \mathbf{h_2'}; v_1, \cdots, v_m)$ of $v$. If this subterm is remained as it was before the reduction step, so (2) is satisfied. Otherwise, it is a subterm of $(\rceil r \lceil^{\mathtt{f}}_{X,Y})\sigma'$, and we have $\mathbf{h_2'} = \sigma'(Y) = \mathbf{h_2}$.

Because of the quasi-interpretation is increasing and the Lemma assumptions,

$$h'_2 = h_2 \geq d \times (\!|\mathtt{f}(\downharpoonright t_1 \downharpoonleft, \cdots, \downharpoonright t_n \downharpoonleft)|\!) \geq d \times (\!|\mathtt{g}(\downharpoonright v_1 \downharpoonleft, \cdots, \downharpoonright v_m \downharpoonleft)|\!)$$

So the condition (ii) is satisfied.

Next, if $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$, we have $\mathbf{h'_1} = \sigma'(Y) = \mathbf{h_2}$. The condition (2) on the definition of a quasi-interpretation enforces that

$$h'_1 = h_2 = d \times (\!|\mathtt{g}(\downharpoonright v_1 \downharpoonleft, \cdots, \downharpoonright v_m \downharpoonleft)|\!) \geq \sum_{i=1}^{m} (\!|\downharpoonright v_i \downharpoonleft|\!)$$

Lastly, if $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ then $h'_1 = h_1 - 1$. Since the rule $e$ is ordered by $MPO'$, there is a permutation $\pi$ such that:

- for all $1 \leq i \leq n$, $v_i \preceq_{mpo'} t_{\pi(i)}$.
- There exists a $j$ such that $v_j \prec_{mpo'} t_{\pi(j)}$.

By Lemma 30, $(\!|\downharpoonright v_i \downharpoonleft|\!) \leq (\!|\downharpoonright t_{\pi(i)} \downharpoonleft|\!)$ for all $1 \leq i \leq n$ and $(\!|\downharpoonright v_j \downharpoonleft|\!) < (\!|\downharpoonright t_{\pi(j)} \downharpoonleft|\!)$. So we have, $h'_1 = h_1 - 1 \geq \sum_{i=1}^{n} (\!|\downharpoonright t_i \downharpoonleft|\!) - 1 \geq \sum_{i=1}^{n} (\!|\downharpoonright v_i \downharpoonleft|\!)$. So (i) is proved. □

**Proposition 36.** *Let main be a $MPO'$-program which admits a quasi interpretation and let $t_1, \cdots, t_n$ be constructor terms of $\mathcal{T}(\mathcal{C})$. Let $d$ be the greatest arity of a function symbol in $\mathcal{F}$. Suppose that $\mathtt{main}(t_1, \cdots, t_n) \xrightarrow{!} s$.*

*Let $\mathtt{main}'$ be the main symbol of $\upharpoonright main \upharpoonleft$. If $h \geq d \times (\!|\mathtt{main}(t_1, \cdots, t_n)|\!)$, we have $\mathtt{main}'(\mathbf{h}, \mathbf{h}; t_1, \cdots, t_n) \xrightarrow{!} s$*

*Proof.* By induction on the length of the derivation and using the above Lemma 35. □

**Theorem 37.** *Let main be a $MPO'$-program which admits a polynomial quasi-interpretation. The function $[\![\mathtt{main}]\!]$ is computed in polynomial time in the size of the inputs.*

*Proof.* The computation of $[\![\mathtt{main}]\!]$ on inputs $t_1, \cdots, t_n$ is performed by evaluating *main* with the $LMPO$-interpreter described in Figure 4.3. The computation runtime is at least as efficient as the evaluation of $\mathtt{main}'(\mathbf{h}, \mathbf{h}; t_1, \cdots, t_n)$ which is defined by the lifted program $\upharpoonright main \upharpoonleft$ and where $h = d \times (\!|\mathtt{main}(t_1, \cdots, t_n)|\!)$. Let $q$ be a polynomial which bounds the quasi-interpretation $(\!|\mathtt{main}|\!)$. So, the size of $h$ is bounded by $q(c \times \max_{i=1}^{i=n} |t_i|)$ for some constant $c$ which depends on constructor interpretations. Following Theorem 25, the runtime of $\mathtt{main}'$ is bounded by a polynomial $p$, in the maximal size of the argument. We conclude, that the runtime of $\mathtt{main}'(\mathbf{h}, \mathbf{h}; t_1, \cdots, t_n)$ is bounded by $p(q(c \times \max_{i=1}^{i=n} |t_i|))$. □

# 7 Conclusion

This paper gives some preliminary directions on how to analyse the implicit complexity of a program and then to generate efficient programs. There are several directions to explore. The role played by $MPO'$ is to encompass algorithmic patterns. For example, a ramified restriction of lexicographic path ordering will permit tail recursion, and divide and conquer algorithms. By representing constructor terms by directed acyclic terms, one might be able to incorporate binary trees. However, the general investigation on implicit complexity of programs depends on advances of the theory of algorithms.

# References


[1] Aehlig, K., and Schwichtenberg, H. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00)* (June 2000). To appear.

[2] Bellantoni, S., and Cook, S. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity 2* (1992), 97–110.

[3] Bellantoni, S., Nigll, K.-H., and Schwichtenberg, H. Ramification, modality, and linearity in higher type recursion. *Annals of pure and applied logic* (2000). To appear.

[4] Benzinger, R. *Automated complexity analysis of NUPRL extracts.* PhD thesis, Cornell University, 1999.

[5] Bonfante, G., Cichon, A., Marion, J.-Y., and Touzet, H. Complexity classes and rewrite systems with polynomial interpretation. In *Computer Science Logic (CSL'98)* (1999), no. 1584 in LNCS, pp. 372–384. Full version will appear in Journal of Functional Programming Languages.

[6] Caseiro, V.-H. An equational characterization of the poly-time functions on any constructor data strucure. Tech. Rep. 226, University of Oslo, Dept. of informatics, December 1996. `http://www.ifi.uio.no/~ftp/publications`.

[7] Cichon, A., and Weiermann, A. Term rewriting therory for the primitive recursive functions. *Annals of pure and applied logic 83* (1997), 199–223.

[8] Clote, P. Computational models and function algebras. In *LCC'94* (1995), D. Leivant, Ed., no. 960 in LNCS, pp. 98–130.

[9] Cobham, A. The intrinsic computational difficulty of functions. In *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, Y. Bar-Hillel, Ed. North-Holland, Amsterdam, 1962, pp. 24–30.

[10] Colson, L. Functions versus algorithms. *Bulletin of EATCS 65* (1998). The logic in computer science column.

[11] Cook, S. Characterizations of pushdown machines in terms of time-bounded computers. *JACM 18(1)* (January 1971), 4–18.

[12] Crary, K., and Weirich, S. Ressource bound certification. In *ACM Symposium on Principles of Programming LanguagesPOPL* (2000). To appear.

[13] Dershowitz, N. Orderings for term-rewriting systems. *Theoretical Computer Science 17*, 3 (1982), 279–301.

[14] Dershowitz, N. Termination of rewriting. *Journal of symbolic computation* (1987), 69–115.

[15] Dershowitz, N., and Jouannaud, J.-P. *Handbook of Theoretical Computer Science vol.B.* Elsevier Science Publishers B. V. (North-Holland), 1990, ch. Rewrite systems, pp. 243–320.

[16] Hofbauer, D. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science 105*, 1 (1992), 129–140.

[17] Hofmann, M. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)* (1999), pp. 464–473.

[18] Hofmann, M. In-place update with linear types or how to compile functional programs into malloc()-free c. In *European Symposium on Programming* (2000).

[19] Hofmann, M. Programming languages capturing complexity classes. *SIGACT News Logic Column 9* (2000).

[20] Huet, G. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery 27*, 4 (October 1980), 797–821.

[21] Jones, N. *Computabilty and complexity from a programming perspective.* MIT press, 1997.

[22] Jones, N. Logspace and ptime characterized by programming languages. *Journal of Theoretical Computer Science 228* (1999), 151–174.

[23] Jones, N. The expressive power of higher order types or, life without cons. Manuscript, 2000.

[24] Kamin, S., and Lévy, J.-J. Attempts for generalising the recursive path orderings. Tech. rep., Univerity of Illinois, Urbana, 1980. Unpublished note.

[25] Lankford, D. On proving term rewriting systems are noetherien. Tech. Rep. MTP-3, Louisiana Technical University, 1979.

[26] Leivant, D. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91)* (1991).

[27] Leivant, D. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, P. Clote and J. Remmel, Eds. Birkhäuser, 1994, pp. 320–343.

[28] Leivant, D. Applicative control and computational complexity. In *Computer Science Logic* (1999).

[29] Marion, J.-Y. Analysing the implicit complexity of programs. Tech. rep., Loria, 2000. Presented at the workshop Implicit Computational Complexity (ICC'99), FLOC, Trento, Italy. The paper is submitted and accessible from `http://www.loria.fr/~marionjy`.

[30] Péter, R. *Rekursive Funktionen.* Akadémiai Kiadó, Budapest, 1966. English translation: *Recursive Functions*, Academic Press, New York, 1967.

[31] Plaisted, D. A recursively defined ordering for proving termination of term rewriting systems. Tech. Rep. R-78-943, Department of Computer Science, University of Illinois, 1978.

[32] Ritchie, R. Classes of predictably computable functions. *Transaction of the American Mathematical Society 106* (1963), 139–173.

[33] Simmons, H. The realm of primitive recursion. *Archive for Mathematical Logic 27* (1988), 177–188.