On Lexicographic Termination Ordering with Space Bound Certifications

G. Bonfante, J.-Y. Marion, and J.-Y. Moyen*

No Institute Given

Abstract. We propose a method to analyse the program space complexity, based on termination orderings. This method can be implemented to certify the runspace of programs. We demonstrate that the class of functions computed by first order functional programs over free algebras which terminate by Lexicographic Path Ordering and admit a polynomial quasi-interpretation, is exactly the class of functions computable in polynomial space.

1 Introduction

Motivations. There are several motivations to develop automatic program complexity analysis :

1. The control of the resources consumed by programs is a necessity, in software development.
2. There is a growing interest in program resource certifications. For example, Benzinger [2] has implemented a prototype to certify the time complexity of programs extracted from Nuprl [6]. Various systems have been defined to control resources in functional languages, see Weirich and Crary [7] and Hofmann [12].
3. Our approach is based on well-known termination orderings, used in term rewriting systems, which are easily implemented.
4. The study of program complexity is of a different nature compared to program verification or termination. It is not enough to know what is computed, we have to know how it is performed. So, it gives rise to interesting questions whose answers might belong to a theory of feasible algorithms which is not yet well established.

Our Results. We consider first order functional programs over any kind of constructors, which terminate by Lexical Path Ordering (LPO). We demonstrate that the class of functions which are computed by LPO-programs admitting polynomially bounded quasi-interpretations, is exactly the class of functions which are computed in polynomial space. (See Section 3.2 for the exact statement.) This resource analysis can be partialy automatized. Indeed, Krishnamoorthy

* Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France. {Guillaume.Bonfante,Jean-Yves.Marion,Jean-Yves.Moyen}@loria.fr

and Narendran in [16] have proved that termination by LPO is NP-complete. To find a quasi-interpretation is not too difficult in general, because the program denotation turns out to be a good candidate.

Complexity and Termination Orderings. Termination orderings give rise to interesting theoretical questions concerning the classes of functions for which they provide termination proofs. Weiermann [23] has shown that LPO characterizes the multiple recursive functions and Hofbauer [10] has shown that Multiset Path Ordering (MPO) gives rise to a characterization of primitive recursive functions. While both of these contain functions which are highly unfeasible, the fact remains that many feasible algorithms can be successfully treated using one or both. Quasi-interpretations allows us to tame the complexity of treated algorithms. Indeed, it has been established [21] that functions computed by programs terminating by MPO and admitting a polynomial quasi-interpretation are exactly the polynomial time computable functions. This last result might be compared with the one of Hofbauer. Analogously, the result presented in this paper might be compared with Weiermann's one.

Others Related Characterizations of Poly-Space There are several characterizations of PSPACE in Finite Model Theory, and we refer to Immerman's book [13] for a complete presentation. A priori, Finite Model Theory approach is not relevant from the point of view of programming languages because computational domains are infinite. But recently, Jones [14] has showed that polynomial space languages are characterized by mean of read-only functional programs. He has observed a closed relationship with a characterization of Goerdt [9].

On infinite computational domains, characterizations of complexity classes go back to Cobham's seminal work [5]. The set of polynomial space computable functions has been identified by Thompson [22]. Those characterizations are based on bounded recursions. Hence, they do not directly study algorithms and so they are not relevant to automatic complexity analysis.

Lastly, from the works of Bellantoni and Cook [1] and of Leivant [18], purely syntactic characterizations of polynomial space computable functions were obtained in [19, 20]. The underlying principle is the concept of ramified recursion. From the point of view of automatic complexity analysis, the main interest of this approach is that we have syntactic criteria to determine the complexity of a program. However, a lot of algorithms are ruled out. Several solutions have been proposed to enlarge the class of algorithms captured. Hofmann [11] has proposed a type system with modalities to deal with non-size increasing functions, e.g. the functions *max* and *min*. Another solution is to introduce a ramified version of termination orderings MPO [20], which delineates polynomial time and polynomial space computable functions.

2 First Order Functional Programming

Throughout the following discussion, we consider three disjoint sets $\mathcal{X}, \mathcal{F}, \mathcal{C}$ of variables, function symbols and constructors.

2.1 Syntax of Programs

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

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

where $x \in \mathcal{X}$, $f \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$[1]*. The size $|t|$ of a term t is the number of symbols in t.*

Definition 2. *A program $\mathcal{E}(\mathtt{main})$ is a set $\mathcal{E}$ of $\mathcal{D}$-rules such that for each rule $f(p_1, \cdots, p_n) \to t$ of $\mathcal{E}$, each variable in t appears also in some pattern p_i. All along the paper, we assume that the set of rules is implicit and we just write `main` to denote the program.*

Example 1. The following program is intended to compute the Ackermann's function. Take the set of constructors to be $\mathcal{C} = \{\mathbf{0}, \mathbf{S}\}$. We won't explicitly note the arity of symbols, as it is implicitly given by the rules.

$$\begin{aligned} \mathtt{Ack}(\mathbf{0}, n) &\to \mathbf{S}(n) \\ \mathtt{Ack}(\mathbf{S}(m), \mathbf{0}) &\to \mathtt{Ack}(m, \mathbf{S}(\mathbf{0})) \\ \mathtt{Ack}(\mathbf{S}(m), \mathbf{S}(n)) &\to \mathtt{Ack}(m, \mathtt{Ack}(\mathbf{S}(m), n)) \end{aligned}$$

2.2 Semantics

The signature $\mathcal{C} \cup \mathcal{F}$ and the set $\mathcal{E}$ of rules induce a rewrite system which brings us the operational semantics. We recall briefly some vocabulary of rewriting theories. For further details, one might consult Dershowitz and Jouannaud's survey [8] from which we take the notations. 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 $\xrightarrow{*}$ is the reflexive-transitive closure of $\to$. Lastly, $t \xrightarrow{!} s$ means that $t \xrightarrow{*} s$ and s is in normal form, *i.e.* no other rule may be applied. A ground (resp. constructor) substitution is a substitution from $\mathcal{X}$ to $\mathcal{T}(\mathcal{C}, \mathcal{F})$ (resp. $\mathcal{T}(\mathcal{C})$).

We now give the semantics of confluent programs, that is programs for which the associated rewrite system is confluent. The domain of interpretation is the constructor algebra $\mathcal{T}(\mathcal{C})$.

Definition 3. *Let `main` be a confluent program. The function computed by `main` is a partial function $[\![\mathtt{main}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ where n is the arity of `main`. For all $u_i \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{main}]\!](u_1, \ldots, u_n) = v$ iff $\mathtt{main}(u_1, \ldots, u_n) \xrightarrow{!} v$ with $v \in \mathcal{T}(\mathcal{C})$. Note that due to the form of the rules a constructor term is a normal form ; as the program is confluent, it is uniquely defined. Otherwise, that is if there is no such normal form, $[\![\mathtt{main}]\!](u_1, \ldots, u_n)$ is undefined.*

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

3 LPO and Quasi-Interpretations

3.1 Lexicographic Path Ordering

Termination orderings are widely used to prove the termination of term rewriting systems. The *Lexicographic Path Ordering* (LPO) is one of them, it was introduced by Kamin and Lévy [15]. We briefly describe it, together with some basic properties we shall use later on.

Definition 4. *Let $\prec$ be a term ordering. We note $\prec^l$ its lexicographic extension. A precedence $\preceq_{\mathcal{F}}$ (strict precedence $\prec_{\mathcal{F}}$) is a quasi-ordering (ordering) on the set $\mathcal{F}$ of function symbols. It is canonically extended on $\mathcal{C} \cup \mathcal{F}$ by saying that constructors are smaller than functions. Given such a precedence, the lexicographic path ordering $\prec_{lpo}$ is defined recursively by the rules:*

$$\frac{s \preceq_{lpo} t_i}{s \prec_{lpo} f(\dots, t_i, \dots)}\ f \in \mathcal{F} \bigcup \mathcal{C} \qquad \frac{s_i \prec_{lpo} f(t_1, \dots, t_n) \qquad g \prec_{\mathcal{F}} f}{g(s_1, \dots, s_m) \prec_{lpo} f(t_1, \dots, t_n)}\ g \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{(s_1, \dots, s_n) \prec^l_{lpo} (t_1, \dots, t_n) \qquad f \approx_{\mathcal{F}} g \qquad s_j \prec_{lpo} f(t_1, \dots, t_n)}{g(s_1, \dots, s_n) \prec_{lpo} f(t_1, \dots, t_n)}$$

Definition 5. *$\lhd$ is the usual subterm ordering. That is $s \lhd f(t_1, \dots, t_n)$ if and only if $s = t_i$ or $s \lhd t_i$ for some $1 \leq i \leq n$.*

Lemma 1. *Let t and s be constructor terms. $s \prec_{lpo} t$ implies $|s| < |t|$.*

Example 2. One can verify that the Ackermann's function of example 1 terminates by LPO.

3.2 Polynomial Quasi-Interpretation

Definition 6. *Let $f \in \mathcal{F} \bigcup \mathcal{C}$ be either a function symbol or a constructor of arity n. A quasi-interpretation of f is a mapping $(\!|f|\!) : \mathbf{N}^n \to \mathbf{N}$ which satisfies (i) $(\!|f|\!)$ is (not necessarily strictly) increasing with respect to each argument, (ii) $(\!|f|\!)(X_1, \dots, X_n) \geq X_i$, for all $1 \leq i \leq n$, (iii) $(\!|f|\!) > 0$ for each 0-ary symbol $f \in \mathcal{F} \cup \mathcal{C}$.*

We extend a quasi-interpretation $(\!|-|\!)$ to terms canonically: $(\!|f(t_1, \dots, t_n)|\!) = (\!|f|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!))$

Definition 7. *$(\!|-|\!)$ is a quasi-interpretation of a program `main` if for each rule $l \to r \in \mathcal{E}($`main`$)$ and for each closed substitution σ, $(\!|r\sigma|\!) \leq (\!|l\sigma|\!)$.*

Lemma 2. *If t and t' are two terms such that $t \to t'$, then $(\!|t|\!) \geq (\!|t'|\!)$.*

Definition 8. *A program `main` admits a polynomial quasi-interpretation $(\!|-|\!)$, if $(\!|-|\!)$ is bounded by a polynomial.*

A polynomial quasi-interpretation is said to be of kind 0 *if for each constructor $\mathbf{c}$, $(\!|\mathbf{c}|\!)(X_1, \dots, X_n) = a + \sum_{i=1}^{n} X_i$ for some constant $a > 0$.*

Remark 1. Quasi-interpretations are not sufficient to prove program termination. Indeed, the rule $\mathtt{f}(x) \to \mathtt{f}(x)$ admits a quasi-interpretation but doesn't terminate.

Unlike quasi-interpretation, a program interpretation satisfies to extra conditions: (i) $(\!| f |\!)(X_1, \ldots, X_n) > X_i$, (ii) $(\!| r\sigma |\!) < (\!| l\sigma |\!)$. Programs admitting an interpretation terminate. This sort of termination proof, by polynomial interpretations, was introduced by Lankford [17]. Bonfante, Cichon, Marion and Touzet [3] proved that programs admitting interpretation of kind 0 are computable in polynomial time.

Definition 9. *A LPO-program is a program that terminates by LPO.*

A $LPO^{Poly(0)}$-program is a LPO-program that admits a quasi-interpretation of kind 0.

Theorem 1 (Main Result). *The set of functions computed by $LPO^{Poly(0)}$-programs is exactly the set of funtions computable in polynomial space.*

Proof. It is a consequence of Theorem 2 and Theorem 4.

Example 3.

1. The Ackermann's function of example 1 doesn't admit a polynomial quasi-interpretation because $[\![\mathtt{Ack}]\!]$ is not polynomially bounded.
2. The Quantified Boolean Formula (QBF) is the problem of the validity of a boolean formula with quantifiers over propositional variables. It is well-known to be PSPACE complete. Wlog, we restrict formulae to $\neg, \vee, \exists$. It can be solved by the following rules:

$$\begin{array}{rlrl}
\mathtt{not}(\mathbf{tt}) \to \mathbf{ff} & & \mathbf{0} = \mathbf{0} \to \mathbf{tt} \\
\mathtt{not}(\mathbf{ff}) \to \mathbf{tt} & & \mathbf{S}(x) = \mathbf{0} \to \mathbf{ff} \\
\mathtt{or}(\mathbf{tt}, x) \to \mathbf{tt} & & \mathbf{0} = \mathbf{S}(y) \to \mathbf{ff} \\
\mathtt{or}(\mathbf{ff}, x) \to x & & \mathbf{S}(x) = \mathbf{S}(y) \to x = y
\end{array}$$

$$\begin{aligned}
\mathtt{main}(\phi) &\to \mathtt{ver}(\phi, \mathbf{nil}) \\
\mathtt{in}(x, \mathbf{nil}) &\to \mathbf{ff} \\
\mathtt{in}(x, \mathbf{cons}(a, l)) &\to \mathtt{or}(x = a, \mathtt{in}(x, l))
\end{aligned}$$

In the next function, t is the set of variables whose value is $\mathbf{tt}$.

$$\begin{aligned}
\mathtt{ver}(\mathbf{Var}(x), t) &\to \mathtt{in}(x, t) \\
\mathtt{ver}(\mathbf{Or}(\phi_1, \phi_2), t) &\to \mathtt{or}(\mathtt{ver}(\phi_1, t), \mathtt{ver}(\phi_2, t)) \\
\mathtt{ver}(\mathbf{Not}(\phi), t) &\to \mathtt{not}(\mathtt{ver}(\phi, t)) \\
\mathtt{ver}(\mathbf{Exists}(n, \phi), t) &\to \mathtt{or}(\mathtt{ver}(\phi, \mathbf{cons}(n, t)), \mathtt{ver}(\phi, t))
\end{aligned}$$

These rules are ordered by LPO by putting $\{\mathtt{not}, \mathtt{or}, \_\mathtt{=}\_\} \prec_{\mathcal{F}} \mathtt{in} \prec_{\mathcal{F}} \mathtt{ver} \prec_{\mathcal{F}} \mathtt{main}$.

They admit the following quasi-interpretations:

 – $(\!| \mathbf{c} |\!)(X_1, \ldots, X_n) = 1 + \sum_{i=1}^{n} X_i$, for each n-ary constructor,
 – $(\!| \mathtt{ver} |\!)(\Phi, T) = \Phi + T$, $(\!| \mathtt{main} |\!)(\Phi) = \Phi + 1$,
 – $(\!| \mathtt{f} |\!)(X_1, \ldots, X_n) = \max_{i=1}^{n} X_i$, for the other function symbols.

4 $\mathbf{LPO}^{\boldsymbol{Poly(0)}}$-Programs are PSPACE Computable

Definition 10. *A* state *is a tuple* $\langle f, t_1, \ldots, t_n \rangle$ *where* f *is a function symbol of arity* n *and* $t_1, \ldots, t_n$ *are constructor terms.*

State(*main*) *is the set of all states built from the symbols of a program* main*.* $State^A(\mathtt{main}) = \{\langle f, t_1, \ldots, t_n \rangle \in State(\mathtt{main}) \ / \ |t_i| < A\}$*. Intutively, a state represents a recursive call in the evaluation process.*

Definition 11. *Let* main *be a* $LPO^{Poly(0)}$*-program,* $\eta_1 = \langle f, t_1, \ldots, t_n \rangle$ *and* $\eta_2 = \langle g, s_1, \ldots, s_m \rangle$ *be two states of State*(main)*. A* transition *is a triplet* $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ *such that:*

(i) $e \equiv f(p_1, \ldots, p_n) \to t$
(ii) there is a substitution σ *such that* $p_i\sigma = t_i$ *for all* $1 \le i \le n$
(iii) there is a subterm $g(u_1, \ldots, u_m) \trianglelefteq t$ *such that* $u_i\sigma \overset{!}{\to} s_i$ *for all* $1 \le i \le n$*.*

$Transition(\mathtt{main})$ *is the set of all transitions between the elements of* $State(\mathtt{main})$*.*

$\overset{*}{\rightsquigarrow}$ *is the reflexive transitive closure of* $\cup_{e \in \mathcal{E}} \overset{e}{\rightsquigarrow}$*.*

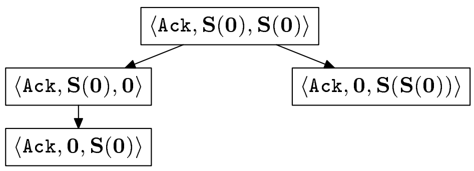
Definition 12. *Let* $\mathcal{E}(\mathtt{main})$ *be a* $LPO^{Poly(0)}$*-program and* $\langle f, t_1, \ldots, t_n \rangle$ *be a state (*i.e. $\langle f, t_1, \ldots, t_n \rangle \in State(\mathtt{main})$*). A* $\langle f, t_1, \ldots, t_n \rangle$*-call tree* τ *is defined as follows:*

- *The root of* τ *is* $\langle f, t_1, \ldots, t_n \rangle$*.*
- *For each node* η_1*, the children of* η_1 *is exactly the set of states* $\{\eta_2 \in State(\mathtt{main}) \ / \ \eta_1 \overset{e}{\rightsquigarrow} \eta_2 \in Transition(\mathtt{main})\}$ *where* e *is a given equation of* $\mathcal{E}$*.*

$CT(\langle f, t_1, \ldots, t_n \rangle)$ *is the set of all* $\langle f, t_1, \ldots, t_n \rangle$*-call trees.*

$$CT^A(\langle f, t_1, \ldots, t_n \rangle) = \{\tau \in CT(\langle f, t_1, \ldots, t_n \rangle) \ / \ \forall \eta \in \tau, \eta \in State^A(\mathtt{main})\}$$

Example 4. The (unique) $\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0}) \rangle$-call tree of $CT(\langle \mathtt{Ack}, \mathbf{S}(\mathbf{0}), \mathbf{S}(\mathbf{0}) \rangle)$ is:



Lemma 3. *Let* main *be a LPO-program,* α *be the number of function symbols in* main *and* d *be the maximal arity of a function symbol. The following facts hold for all* $\tau \in CT^A(\langle f, t_1, \ldots, t_n \rangle)$*:*

1. *If* $\langle f, t_1, \ldots, t_n \rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \ldots, s_m \rangle$ *then (a)* $g \prec_{\mathcal{F}} f$ *or (b)* $g \approx_{\mathcal{F}} f$ *and* $(s_1, \ldots, s_m) \prec^l_{lpo} (t_1, \ldots, t_n)$*.*
2. *If* $\langle f, t_1, \ldots, t_n \rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \ldots, s_m \rangle$ *in* τ *and* $g \approx_{\mathcal{F}} f$ *then the number of states between* $\langle f, t_1, \ldots, t_n \rangle$ *and* $\langle g, s_1, \ldots, s_m \rangle$ *is bounded by* A^d*.*

3. The length of each branch of τ is bounded by $\alpha \times A^d$.

Proof.

1. Because the rules of the program decrease for LPO.
2. Let $\langle \mathtt{h}, u_1, \dots, u_p \rangle$ be a child of $\langle \mathtt{f}, t_1, \dots, t_n \rangle$. As t_i and u_j are constructor terms, due to first point of the current lemma and lemma 1, we have $(|u_1|, \cdots, |u_p|) <^l (|t_1|, \cdots, |t_n|)$. Since the size of each component is bounded by A, the length of the decreasing chain is bounded by A^d.
3. In each branch, there are at most A^d states whose function symbols have the same precedence, then A^d states whose function symbol have the precedence immediatly below, and so on. As there are only α function symbols the length of the branch is bounded by $\alpha \times A^d$.

Lemma 4. *Let* `main` *be a $LPO^{Poly(0)}$-program, f be a function symbol and $t_1, \dots, t_n$ be constructor terms. $CT(f, t_1, \dots, t_n) = CT^{(\!|f(t_1,\dots,t_n)|\!)}(f, t_1, \dots, t_n)$.*

Proof. Let $\tau \in CT(\mathtt{f}, t_1, \dots, t_n)$ and $\langle \mathtt{g}, s_1, \dots, s_m \rangle$ be a state in τ. As s_i is a constructor term, $|s_i| \leq (\!|s_i|\!) \leq (\!|\mathtt{g}(s_1, \dots, s_m)|\!) \leq (\!|\mathtt{f}(t_1, \dots, t_n)|\!)$ because of the definition of quasi-interpretations.

Lemma 5. *Given a term $t \in \mathcal{T}(\mathcal{C})$, the following holds: $(\!|t|\!) \leq c.|t|$ for some constant c. As a corollary, we have $(\!|main(t_1, \dots, t_n)|\!) \leq P(\max_{i=1}^n |t_i|)$ for some polynomial P.*

Theorem 2. *Let* `main` *be a $LPO^{Poly(0)}$-program. For each constructor terms $t_1, \dots, t_n$, the space used by a call by value interpreter to compute* `main`$(t_1, \dots, t_n)$ *is bounded by a polynomial in $max_i\{|t_i|\}$. Such an interpreter is described in annex.*

Proof. Put $A = (\!|\mathtt{main}(t_1, \dots, t_n)|\!)$.

The interpreter only needs to store the call stack of each recursive call and the intermediate terms (e_i, b) of the computation. The size of e_i and b are both bounded by A. Note that the computation can be followed on a $\langle \mathtt{main}, t_1, \dots, t_n \rangle$ call-tree. Each recursive call corresponds a transition on the call-tree. So, the maximal depth of the stack corresponds to the maximal length of the branch in a call tree of $CT\langle \mathtt{main}, t_1, \dots, t_n \rangle = CT^A\langle \mathtt{main}, t_1, \dots, t_n \rangle$. So it is bounded by $\alpha \times A^d$ (see Lemma 3(3)). The values stored in the stack are states; as a consequence, the size of each of them is bounded by $d \times A + O(1)$.

Therefore, the space used by the interpreter is bounded by $\alpha \times d \times A^{d+1} + A + O(1)$, and A is a polynomial in the size of $\max_i\{|t_i|\}$ by Lemma 5.

5 Parallel Register Machines over Words

We present here a restriction of Parallel Register Machines introduced in [19]. They are an adaptation of the alternating Turing machine. Chandra, Kozen and Stockmeyer have demonstrated that the set of functions that alternating Turing machines computes in polynomial time is exactly the state of polynomial space computable functions.

Definition 13. $\mathbb{W} = \mathcal{T}(\{\boldsymbol{0}^1, \boldsymbol{1}^1, \boldsymbol{\epsilon}^0\})$. $\mathbb{N} = \mathcal{T}(\{\boldsymbol{s}^1, \diamond^0\})$.

Note that $\mathbb{W}$ *(resp.* $\mathbb{N}$*) is isomorphic to* $\{\boldsymbol{0}, \boldsymbol{1}\}^*$ *(resp. natural numbers), both sets are used indiferently in the rest of the section.*

Definition 14. *A Parallel Register Machine (PRM) over the word algebra* $\mathbb{W}$ *consists in:*

1. *a finite set* $S = \{s_0, s_1, \ldots, s_k\}$ *of* states*, including a distinct state* BEGIN.
2. *a finite list* $\Pi = \{\pi_1, \ldots, \pi_m\}$ *of* registers*; we write* OUTPUT *for* π_m*; Registers will only store values in* $\mathbb{W}$*;*
3. *an ordering* $\blacktriangleleft$ *on* $\mathbb{W}$*:* $\boldsymbol{\epsilon} \blacktriangleleft y$, $\boldsymbol{0}(x) \blacktriangleleft \boldsymbol{1}(y)$, $\boldsymbol{i}(x) \blacktriangleleft \boldsymbol{i}(y)$ *if and only if* $x \blacktriangleleft y$.
4. *a function com mapping states to* commands *which are :*$[\boldsymbol{Succ}(\pi' = \boldsymbol{i}(\pi), s')]$, $[\boldsymbol{Pred}(\pi' = \mathbf{p}(\pi), s')]$, $[\boldsymbol{Branch}(\pi, s', s'')]$, $[\boldsymbol{Fork}_{min}(s', s'')]$, $[\boldsymbol{Fork}_{max}(s', s'')]$, $[\boldsymbol{End}]$.

A *configuration* of a PRM M is given by a pair (s, F) where $s \in S$ and F is a function $\Pi \to \mathbb{W}$. We note $[u_1, \ldots, u_m]$ for the function which maps $\pi_i \mapsto u_i$ and $\{\pi_i \leftarrow a\}[u_1, \ldots, u_m]$ denotes $[u_1, \ldots, u_{i-1}, a, u_{i+1}, \ldots, u_m]$.

Definition 15. *Given* M *as above we define a semantic partial-function* $\mathtt{eval} : \mathbb{N} \times S \times \mathbb{W}^m \mapsto \mathbb{W}$*, that maps the result of the machine in a "time bound" given by the first argument.*

- $\mathtt{eval}(0, s, F)$ *is undefined.*
- *If* $com(s)$ *is* $\boldsymbol{Succ}(\pi' = \boldsymbol{i}(\pi), s')$ *then* $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', \{\pi' \leftarrow \boldsymbol{i}(\pi)\}F)$. *Note that on the right of the left arrow,* π *denotes the content of the register;*
- *If* $com(s)$ *is* $\boldsymbol{Pred}(\pi' = \mathbf{p}(\pi), s')]$, *then* $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', \{\pi' \leftarrow \mathbf{p}(\pi)\}F)$*;*
- *If* $com(s)$ *is* $\boldsymbol{Branch}(\pi, s', s'')$ *then* $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, r, F)$*, where* $r = s'$ *if* $\pi = \boldsymbol{0}(w)$ *and* $r = s''$ *if* $\pi = \boldsymbol{1}(w)$*;*
- *If* $com(s)$ *is* $\boldsymbol{Fork}_{\min}(s', s'')$ *then* $\mathtt{eval}(t+1, s, F) = \min_{\blacktriangleleft}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$*;*
- *If* $com(s)$ *is* $\boldsymbol{Fork}_{max}(s', s'')$ *then* $\mathtt{eval}(t+1, s, F) = \max_{\blacktriangleleft}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$*;*
- *If* $com(s)$ *is* $\boldsymbol{End}$ *then* $\mathtt{eval}(t+1, s, F) = F(\text{OUTPUT})$.

Definition 16. *Given a function* $T : \mathbb{N} \to \mathbb{N}$*, we say that the PRM* M *computes* $f : \mathbb{W}^k \to \mathbb{W}$ *in time* T *(or equivalently that* f *is* T*-computable) if for all* $(w_1, \ldots, w_k) \in \mathbb{W}^k$*, we have*

$$\mathtt{eval}(T(\max_{i=1}^{k} |w_i|), \text{BEGIN}, [w_1, \ldots, w_k, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon}]) = f(w_1, \ldots, w_k)$$

Theorem 3 (Chandra & *al* [4]). *Let* $f : \mathbb{W} \to \mathbb{W}$. f *is computable in polynomial space iff* f *is PRM-computable in polynomial time.*

5.1 Simulation of PRMs by LPO$^{Poly(0)}$-Programs

The simulation of PRM is done simply by following the rules of the operational semantics of PRM we gave above. In particular, the first argument of `eval` represents a clock.

Lemma 6 (Plug and Play Lemma). *Let $f : \mathbb{W} \to \mathbb{W}$ be a T-time PRM-computable function, then, the function f' is computable by an LPO$^{Poly(0)}$.*

$$\begin{array}{rll} f' : \mathbb{N} \times \mathbb{W} \to \mathbb{W} & & \\ (n, w) \mapsto f(w) & \text{if } n > T(|w|) \\ (n, w) \mapsto \bot & \text{otherwise} \end{array}$$

Proof. Let the set of constructors be $\mathcal{C} = \{\mathbf{0}, \mathbf{1}, \mathbf{s}, \diamond, \epsilon\} \cup \mathcal{S}$ where S is the set of states. We let the rules in appendix B. Simply, let's say that the functions symbols are: $\mathtt{min}, \mathtt{max}$ corresponding to $\min_{\blacktriangleleft}, \max_{\blacktriangleleft}$, `eval` which simulate the rules of the operational semantics and $\mathtt{f}'$. We develop here two rules for `eval`.

- $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \dots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \dots, \pi_m)$ if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,
- $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \pi_m)$
  $\to \mathtt{min}(\mathtt{Eval}(t, s', \pi_1, \dots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \dots, \pi_m))$ if $com(s) = \mathbf{Fork}_{\min}(s', s'')$

Take the precedence $\{\mathtt{min}, \mathtt{max}\} \prec_{\mathcal{F}} \mathtt{Eval}$, these rules decrease according to LPO because the time bound decrease. They admit the following quasi-interpretation:

$$\begin{array}{lll} (\!|\epsilon|\!) = 1 & (\!|\mathbf{0}|\!)(X) = X + 1 & (\!|\mathtt{min}|\!)(W, W') = \max(W, W') \\ (\!|\diamond|\!) = 1 & (\!|\mathbf{1}|\!)(X) = X + 1 & (\!|\mathtt{max}|\!)(W, W') = \max(W, W') \\ & (\!|\mathbf{s}|\!)(X) = X + 1 & \end{array}$$

$$\forall s \in S, (\!|s|\!) = 1 \qquad (\!|\mathtt{Eval}|\!)(T, S, \Pi_1, \dots, \Pi_m) = \max\{\Pi_1, \dots, \Pi_m\} \times T + S$$

Now, the function $\mathtt{f}'$ is defined by $\mathtt{f}'(n, w) \to \mathtt{Eval}(n, \textsc{begin}, w, \boldsymbol{\epsilon}, \dots, \boldsymbol{\epsilon})$. It is routine to check that $f' = [\![\mathtt{f}']\!]$. This rule decreases by LPO by taking $\mathtt{f}' \succ \mathtt{Eval}$. There is also a quasi-interpretation for the rule: $(\!|\mathtt{f}'|\!)(N, X) = N \times X + 1$.

Lemma 7. *Given $w \in \mathbb{W}$ and a polynomial P, the function $w \in \mathbb{W} \mapsto P(|w|)$ is computable by a LPO$^{Poly(0)}$-program. The proof is given in the appendix.*

Theorem 4. *A polynomial time PRM computable function f can be computed by an LPO$^{Poly(0)}$-program.*

Proof. Follows from Lemma 6 and Lemma 7.

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.
2. R. Benzinger. *Automated complexity analysis of NUPRL extracts.* PhD thesis, Cornell University, 1999.
3. G. Bonfante, A. Cichon, J-Y Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Computer Science Logic, 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384, 1999.
4. A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.
5. A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.
6. R. Constable and al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.
7. K. Crary and S. Weirich. Ressource bound certification. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL*, pages 184 – 198, 2000.
8. N. Dershowitz and J-P Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. Elsevier Science Publishers B. V. (NorthHolland), 1990.
9. A. Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100(1):45–66, 1992.
10. D. Hofbauer. Termination proofs with multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.
11. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.
12. M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.
13. N. Immerman. *Descriptive Complexity.* Springer, 1999.
14. N. Jones. The Expressive Power of Higher order Types or, Life without CONS. to appear, 2000.
15. S. Kamin and J-J Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerity of Illinois, Urbana, 1980. Unpublished note.
16. M. S. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theoretical Computer Science*, 40(2-3):323–328, October 1985.
17. D.S. Lankford. On proving term rewriting systems are noetherien. Technical Report MTP-3, Louisiana Technical University, 1979.
18. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.
19. D. Leivant and J-Y Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500, Kazimierz,Poland, 1995. Springer.

20. J-Y Marion. Complexité implicite des calculs, de la théorie à la pratique, 2000. Habilitation.
21. J-Y Marion and J-Y Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.
22. D.B. Thompson. Subrecursiveness : machine independent notions of computability in restricted time and storage. *Math. System Theory*, 6:3–15, 1972.
23. A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.

A Interpreter for LPO$^{Poly(0)}$-Programs

A space-economical interpreter:

function eval($\mathtt{f}, t_1, \ldots, t_n$)
begin
  let $e = \mathtt{f}(p_1, \ldots, p_n) \to t$
  let σ such that $p_i\sigma = t_i$
  let $e_0 = t\sigma$
  let $i = 0$
  foreach $\mathtt{g}(s_1, \ldots, s_m) \trianglelefteq e_i \wedge s_j \in \mathcal{T}(\mathcal{C}), j \in \{1, .., m\}$ **do**
    $b = \mathrm{eval}(\mathtt{g}, s_1, \ldots, s_m)$
    $e_{i+1} = e_i\{\mathtt{g}(s_1, \ldots, s_m) \leftarrow b\}$
    $i = i + 1$
  return e_i
end.

where $e_i\{\mathtt{g}(s_1, \ldots, s_m) \leftarrow b\}$ denotes the term e_i where each occurence of $\mathtt{g}(s_1, \ldots, s_m)$ has been replaced by b.

B Simulation of PRM by LPO$^{Poly(0)}$-Programs

One follows the operational semantics of PRMs.

$$
\begin{array}{rlrl}
\mathtt{min}(\epsilon, w) \to \epsilon & & \mathtt{max}(\epsilon, w) \to w \\
\mathtt{min}(w, \epsilon) \to \epsilon & & \mathtt{max}(w, \epsilon) \to w \\
\mathtt{min}(\mathbf{0}(w), \mathbf{1}(w')) \to \mathbf{0}(w) & & \mathtt{max}(\mathbf{0}(w), \mathbf{1}(w')) \to \mathbf{1}(w') \\
\mathtt{min}(\mathbf{1}(w), \mathbf{0}(w')) \to \mathbf{0}(w') & & \mathtt{max}(\mathbf{1}(w), \mathbf{0}(w')) \to \mathbf{1}(w) \\
\mathtt{min}(\mathbf{i}(w), \mathbf{i}(w')) \to \mathbf{i}(\mathtt{min}(w, w')) & & \mathtt{max}(\mathbf{i}(w), \mathbf{i}(w')) \to \mathbf{i}(\mathtt{max}(w, w'))
\end{array}
$$

with $\mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}$. We have $[\![\mathtt{min}]\!] = \min_{\blacktriangleleft}$ and $[\![\mathtt{max}]\!] = \max_{\blacktriangleleft}$.

(a) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \ldots, \pi_{j-1}, \mathbf{i}(\pi_k), \pi_{j+1}, \ldots, \pi_m)$
 if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_k), s')$,

(b) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \mathbf{i}(\pi'_j), \ldots, \pi_m)$
 $\to \mathtt{Eval}(t, s', \pi_1, \ldots, \pi'_j, \ldots, \mathbf{i}(\pi'_j), \ldots, \pi_m)$
 if $com(s) = \mathbf{Pred}(\pi_k = \mathbf{p}(\pi_j), s')$,

(c) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_{j-1}, \mathbf{i}(\pi_j), \pi_{j+1}, \ldots, \pi_m) \to \mathtt{Eval}(t, r, \pi_1, \ldots, \pi_m)$
 if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$ where $r = s'$ if $\mathbf{i} = \mathbf{0}$ and $r = s''$ if $\mathbf{i} = \mathbf{1}$,

(d) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m)$
 $\to \mathtt{min}(\mathtt{Eval}(t, s', \pi_1, \ldots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
 if $com(s) = \mathbf{Fork}_{\min}(s', s'')$

(e) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m)$
$\quad\quad\rightarrow \texttt{max}(\texttt{Eval}(t, s', \pi_1, \ldots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\max}(s', s'')$

(f) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \rightarrow \pi_m$
if $com(s) = \mathbf{End}$

B.1 Computation of Polynomials by LPO$^{Poly(0)}$-Programs

Each polynomials can be computed with a combination of `add` and `mult`.

$$\begin{array}{rlrl} \texttt{add}(\diamond, y) & \rightarrow y & \texttt{mult}(\diamond, y) & \rightarrow \diamond \\ \texttt{add}(\mathbf{s}(x), y) & \rightarrow \mathbf{s}(\texttt{add}(x, y)) & \texttt{mult}(\mathbf{s}(x), y) & \rightarrow \texttt{add}(y, \texttt{mult}(x, y)) \end{array}$$

These functions are clearly ordered by LPO by putting $\texttt{add} \prec_{\mathcal{F}} \texttt{mult}$. And they admit the quasi-interpretations:

$$(\!|\texttt{add}|\!)(X, Y) = X + Y$$
$$(\!|\texttt{mult}|\!)(X, Y) = X \times Y$$

Remark 2. The interpretation of a polynomial correponds to its semantics.