Quasi-interpretations

G. Bonfante [a] J.-Y. Marion [a] J.-Y. Moyen [b]

[a] *Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, and École Nationale Supérieure des Mines de Nancy, INPL, France.*

[b] *Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, and Université Henri Poincaré Nancy I, France.*

Abstract

This paper presents in a reasoned way our works on resource analysis by quasi-interpretations. The controlled resources are typically the runtime, the runspace or the size of a result in a program execution.

Quasi-interpretations assign to each program symbol a numerical function which is compatible with the computationnal semantics. The quasi-interpretation method offers several advantages. It allows to predict system complexity, may provide hints in order to optimize the execution, it gives resource certificates, and finally, can be automated. We propose a method to determine if a program admits or not a quasi-interpretation in a broad class which is relevant for feasible computations. By combining the quasi-interpretation method with termination tools (here term orderings), we have obtained several characterizations of complexity classes starting from Ptime and Pspace.

1 Introduction

This paper is part of a general investigation on program complexity analysis. We present the quasi-interpretation method which applies to any formalism that can be reduced to transition systems. A quasi-interpretation gives a kind of measures by assigning to each symbol of a system a monotonic numerical function over $\mathbb{R}^+$. A quasi-interpretation possesses two main properties. First, the quasi-interpretation of a ground term is a real which bounds its size. Second, a quasi-interpretation weakly decreases when a term is reduced.

Email addresses: bonfante@loria.fr (G. Bonfante), marionjy@loria.fr (J.-Y. Marion), moyen@loria.fr (J.-Y. Moyen).

From a practical point of view, the bottom line is this. The quasi-interpretation method is a tool to perform complexity analysis in a static way. Quasi-interpretations allow to establish an upper bound on the size of intermediate values which occur in a computation. This was used for a resource byte-code verifier in [2]. Moreover in the context of mobile-code or of secured application, a resource certificate can be sent.

We restrict our study to quasi-interpretations over $\mathbb{R}^+$ which are bounded by some polynomials. A consequence of Tarski's Theorem [37] is that it is decidable whether or not a program admits a quasi-interpretation. This leads to an automatic synthesis procedure.

From a theoretical point of view, we combine quasi-interpretations with termination tools. We focus on simplification orderings and we consider in particular Recursive Path Orderings introduced by Dershowitz [16]. It turns out that we characterize the class Ptime of functions computable in polynomial time and the class Pspace of functions computable in polynomial space.

This work is related to Cobham [11], Bellantoni and Cook [4], Leivant [26] and Marion [27] ideas to delineate complexity classes. Note that most of the machine-independent characterizations of complexity classes have an extensional point of view. They study functions and do not pay too much attention to the algorithmic aspects. In this paper, we try an alternative way of looking at complexity classes by focusing on algorithms, that is the way things are computed rather than what is computed. In this long-term research program, the completeness problematic has moved and the nature of the problem has changed. Indeed, the class of algorithms (with respect to some encoding), say which run in polynomial time, is not recursively enumerable. So we cannot expect to characterize all Ptime algorithms. But we think that this question could shed light on the nature of computations and contribute to intentional computability theory. Similar interrogations have been brought up by Caseiro [8], Hofmann [20] and Jones [22].

The paper organization is the following. The next Section introduces the first order functional programming language. The quasi-interpretations are defined next in Section 3. We suggest a classification of quasi-interpretations which induces a natural complexity hierarchy. Then, we study quasi-interpretation properties. Section 4 establishes that it is decidable if a program admits a quasi-interpretation wrt a broad class of polynomially bounded assignments. Section 5 defines recursive path orderings used to prove termination of programs and some properties that we shall use later on. After these three sections, we state the main results at the beginning of Sections 6 and 7. Roughly speaking, the first result says that programs which terminate by product or lexicographic orderings are computable in polynomial-space. The second result means that programs that terminate by product ordering or that are tail

recursive are computable in polynomial time. It is worth noticing that we have to compute the program by call by value semantics with a cache to have an exponential speed-up. The last Section 8 is devoted to simulations.

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 and the rules are defined in the following way:*

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

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$. We shall use a type writer font for function symbols and a bold face font for constructors.

Definition 2 *A program is a quadruplet $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ 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. We distinguish among $\mathcal{F}$ a main function symbol whose name is given by the program name $\mathtt{f}$.*

Throughout, we consider orthogonal programs which is a sufficient condition in order to be confluent. Following Huet [21], the program rules satisfy both conditions: (i) Each rule $\mathtt{f}(p_1, \cdots, p_n) \to t$ is left-linear, that is a variable appears only once in $\mathtt{f}(p_1, \cdots, p_n)$, and (ii) there are no two left hand-sides which are overlapping.

The size $|t|$ of a term t is defined by $|b| = 0$ and $|b(t_1, \ldots, t_n)| = 1 + \sum_i |t_i|$ where $b \in \mathcal{C} \cup \mathcal{F}$.

2.2 Semantics

The domain of the computed functions is the constructor algebra $\mathcal{T}(\mathcal{C})$. A substitution σ is a mapping from variables to terms. We say that it is a

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(v_1, \cdots, v_n)} \textit{ (Constructor)}$$

$$\frac{t_i \downarrow v_i \quad \mathtt{f}(p_1, \cdots, p_n) \rightarrow r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\mathtt{f}(t_1, \cdots, t_n) \downarrow v} \textit{ (Function)}$$

Fig. 1. Call by value semantics with respect to a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$

constructor substitution when the range of σ is $\mathcal{T}(\mathcal{C})$. We note $\mathfrak{S}$ the set of these constructor substitutions.

We consider a call by value semantics which is displayed in Figure 1. The meaning of $t \downarrow v$ is that t evaluates to the constructor term v. The program $\mathtt{f}$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \rightarrow \mathcal{T}(\mathcal{C})$ defined as follows. For all $u_i \in \mathcal{T}(\mathcal{C}), [\![\mathtt{f}]\!](u_1, \cdots, u_n) = v$ iff $\mathtt{f}(u_1, \cdots, u_n) \downarrow v$. Otherwise $[\![\mathtt{f}]\!](u_1, \cdots, u_n)$ is undefined.

3 Quasi-interpretations

To approach the resource control problem, we suggest the concept of quasi-interpretation which plays the main role in this study. Quasi-interpretations have been introduced by Bonfante [5], Marion [30,31] and Marion-Moyen [32].

The set of non-negative real numbers is noted $\mathbb{R}^+$.

Definition 3 (Assignment) *An* assignment of a symbol $b \in \mathcal{F} \bigcup \mathcal{C}$ whose arity is n is a function $(\!|b|\!) : (\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$ such that:

(Subterm) $(\!|b|\!)(X_1, \cdots, X_n) \geq X_i$ for all $1 \leq i \leq n$.
(Weak Monotonicity) $(\!|b|\!)$ is increasing (not necessarily strictly) with respect to each variable.

We extend assignments $(\!|-|\!)$ to terms canonically. Given a term t with n variables, the assignment $(\!|t|\!)$ is a function $(\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$ defined by the rules:

$$(\!|b(t_1, \cdots, t_n)|\!) = (\!|b|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!))$$
$$(\!|x|\!) = X$$

Given two functions $f : (\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$ and $g : (\mathbb{R}^+)^m \rightarrow \mathbb{R}^+$ such that $n \geq m$, we say that $f \geq g$ iff $\forall X_1, \ldots, X_n : f(X_1, \ldots, X_n) \geq g(X_1, \ldots, X_m)$.

There are some well-known and useful consequences of such definitions. We have $(\!|s|\!) \geq (\!|t|\!)$ if t is a subterm of s. Then, for every substitution σ, $(\!|s|\!) \geq (\!|t|\!)$ implies that $(\!|s\sigma|\!) \geq (\!|t\sigma|\!)$.

Definition 4 (Quasi-interpretation) *A program assignment $(\!|-|\!)$ is an assignment of each program symbol. An assignment $(\!|-|\!)$ of a program is a quasi-interpretation if for each rule $l \to r$,*

$$(\!|l|\!) \geq (\!|r|\!)$$

It is worth noticing that the above inequality is not strict which differs from the notion of interpretation used to prove termination that we briefly present in Section 3.6.

Example 5

Given a list l of tally natural numbers, $\mathtt{sort}(l)$ sorts the elements of l by insertion. The constructor set is $\mathcal{C} = \{\mathbf{tt}, \mathbf{ff}, \mathbf{0}, \mathbf{S}, \mathbf{nil}, \mathbf{cons}\}$.

$$\begin{aligned}
\mathtt{if}\ \ \mathbf{tt}\ \mathtt{then}\ \ x\ \mathtt{else}\ \ y &\to x \\
\mathtt{if}\ \ \mathbf{ff}\ \mathtt{then}\ \ x\ \mathtt{else}\ \ y &\to y \\
\mathbf{0} < \mathbf{S}(y) &\to \mathbf{tt} \\
x < \mathbf{0} &\to \mathbf{ff} \\
\mathbf{S}(x) < \mathbf{S}(y) &\to x < y \\
\mathtt{insert}(a, \mathbf{nil}) &\to \mathbf{cons}(a, \mathbf{nil}) \\
\mathtt{insert}(a, \mathbf{cons}(b, l)) &\to \mathtt{if}\ \ a < b\ \mathtt{then}\ \ \mathbf{cons}(a, \mathbf{cons}(b, l)) \\
&\qquad\qquad \mathtt{else}\ \ \mathbf{cons}(b, \mathtt{insert}(a, l)) \\
\mathtt{sort}(\mathbf{nil}) &\to \mathbf{nil} \\
\mathtt{sort}(\mathbf{cons}(a, l)) &\to \mathtt{insert}(a, \mathtt{sort}(l))
\end{aligned}$$

It admits the following quasi-interpretation.

- $(\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) = (\!|\mathbf{0}|\!) = (\!|\mathbf{nil}|\!) = 0$
- $(\!|\mathtt{if\ \ then\ \ else}|\!)(X, Y, Z) = \max(X, Y, Z)$
- $(\!|<|\!)(X, Y) = \max(X, Y)$
- $(\!|\mathbf{S}|\!)(X) = X + 1$
- $(\!|\mathbf{cons}|\!)(X, Y) = (\!|\mathtt{insert}|\!)(X, Y) = X + Y + 1$
- $(\!|\mathtt{sort}|\!)(X) = X$

This example illustrates two important facts. Quasi-interpretations can be max-functions like in the case of $<$. The quasi-interpretations of both sides of a rule can be the same. For example take the last rule. We see that

$$(\!|\mathtt{sort}(\mathbf{cons}(a, l))|\!) = A + L + 1 = (\!|\mathtt{insert}(a, \mathtt{sort}(l))|\!)$$

Example 6

Given two binary words u and v over the constructor set $\{\mathbf{a}, \mathbf{b}, \boldsymbol{\epsilon}\}$, $\mathtt{lcs}(u, v)$ returns the the length of the longest common subsequence of u and v. The expression $\mathtt{lcs}(\mathbf{ababa}, \mathbf{baaba})$ evaluates to $\mathbf{S}^4(\mathbf{0})$ because the length longest common subsequence is 4 (take **baba**).

$$\begin{array}{rll}
\mathtt{lcs}(\boldsymbol{\epsilon}, y) & \to \mathbf{0} & \\
\mathtt{lcs}(x, \boldsymbol{\epsilon}) & \to \mathbf{0} & \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y)) & \to \mathbf{S}(\mathtt{lcs}(x, y)) & \mathbf{i}, \mathbf{j} \in \{\mathbf{a}, \mathbf{b}\} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y)) & \to \mathtt{max}(\mathtt{lcs}(x, \mathbf{j}(y)), \mathtt{lcs}(\mathbf{i}(x), y)) & \mathbf{i} \neq \mathbf{j} \\
\mathtt{max}(n, \mathbf{0}) & \to n & \\
\mathtt{max}(\mathbf{0}, m) & \to m & \\
\mathtt{max}(\mathbf{S}(n), \mathbf{S}(m)) & \to \mathbf{S}(\mathtt{max}(n, m)) &
\end{array}$$

It admits the following quasi-interpretation:

- $(\!|\boldsymbol{\epsilon}|\!) = (\!|\mathbf{0}|\!) = 1$
- $(\!|\mathbf{a}|\!)(X) = (\!|\mathbf{b}|\!)(X) = (\!|\mathbf{S}|\!)(X) = X + 1$
- $(\!|\mathtt{lcs}|\!)(X, Y) = (\!|\mathtt{max}|\!)(X, Y) = \max(X, Y)$

3.1 Taxonomy of Quasi-interpretations

We shall henceforth consider quasi-interpretations which are bounded by polynomials. Other classes of assignments could be introduced such as elementary or primitive recursive assignments, but we will not discuss about them in this paper. This type of extensions is related to Lescanne's paper [29] about interpretation for termination proofs.

Definition 7 *An assignment $(\!|-|\!)$ is polynomial if for each symbol $b \in \mathcal{F} \bigcup \mathcal{C}$, $(\!|b|\!)$ is a function bounded by a polynomial. A quasi-interpretation $(\!|-|\!)$ is polynomial if the assignment $(\!|-|\!)$ is polynomial.*

Next, we classify polynomial quasi-interpretations according to the rate of growth of constructor assignments.

Definition 8 *Let* $\mathbf{c}$ *be a constructor of arity $n > 0$.*

- *An assignment of* $\mathbf{c}$ *is* additive *(or of kind 0) if*

$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha \qquad \alpha \geq 1$$

- *An assignment of* $\mathbf{c}$ *is* affine *(or of kind 1) if*

$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = P(X_1, \cdots, X_n) + \alpha \qquad \alpha \geq 1$$

  where P *is a polynomial whose degree is at most* 1 *in each variable.*
- *An assignment* $\mathbf{c}$ *is* multiplicative *(or of kind 2) if*

$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = Q(X_1, \cdots, X_n) + \alpha \qquad \alpha \geq 1$$

  where Q *is any polynomial.*

We classify *polynomial* quasi-interpretations by the kind of assignment given to constructors.

- If each constructor assignment is additive then the quasi-interpretation is additive.
- If each constructor assignment is affine then the quasi-interpretation affine.
- If each constructor assignment is multiplicative then the quasi-interpretation multiplicative.

This program classification is concerned with the kind of quasi-interpretation given to constructors and not to function symbols. In example 5, the program admits an additive quasi-interpretation because each constructor (that is the symbol in $\{\mathbf{tt}, \mathbf{ff}, \mathbf{0}, \mathbf{S}, \mathbf{nil}, \mathbf{cons}\}$) admits an additive assignment. On the other hand, the assignment of the function symbol $<$ is not additive but it is does not matter because it is not a constructor. For the same reason, the `lcs` example admits also an additive quasi-interpretation.

All along, it is convenient to just write "additive (resp.affine, multiplicative) program" instead of a "program which admits an additive (resp.affine, multiplicative) quasi-interpretation".

3.2 Elementary properties of Quasi-interpretations

Proposition 9 *Suppose that* t *is a constructor term in* $\mathcal{T}(\mathcal{C})$.

- *For an additive program, we have* $(\!|t|\!) \leq O(|t|)$
- *For an affine program, we have* $(\!|t|\!) \leq 2^{O(|t|)}$
- *For a multiplicative program, we have* $(\!|t|\!) \leq 2^{2^{O(|t|)}}$

PROOF. The proof goes by induction on the size of t. It is written in [7]. □

It is worth noticing that the above Proposition illustrates a general phenomenon that we shall see all along this paper. Roughly speaking, the complexity increases by an exponential when we jump from additive to affine quasi-interpretations, or from affine to multiplicative ones.

Proposition 10 *Suppose that t is a constructor term in $\mathcal{T}(\mathcal{C})$.*
We have $|s| \leq (\!|s|\!)$.

PROOF. The proof goes by induction on the size of t. □

3.3 Call-trees

We present now call-trees which are a tool that we shall use all along. Let $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. A call-tree gives a static view of an execution and captures all function calls. Hence, we can study dependencies between function calls without taking care of the extra details provided by the underlying rewriting relation.

Definition 11 *A* state *is a tuple $\langle \mathtt{h}, u_1, \cdots, u_p \rangle$ where $\mathtt{h}$ is a function symbol of arity p and $u_1, \ldots, u_p$ are constructor terms. Assume that $\eta_1 = \langle \mathtt{h}, u_1, \cdots, u_p \rangle$ and $\eta_2 = \langle \mathtt{g}, s_1, \cdots, s_m \rangle$ are two states. A* transition *is a triplet $\eta_1 \stackrel{e}{\leadsto} \eta_2$ such that:*

(i) e is a rule $\mathtt{h}(q_1, \cdots, q_p) \to t$ of $\mathcal{E}$,
(ii) there is a substitution σ such that $q_i\sigma = u_i$ for all $1 \leq i \leq p$,
(iii) there is a subterm $\mathtt{g}(v_1, \cdots, v_m)$ of t such that $v_i\sigma \downarrow s_i$ for all $1 \leq i \leq m$.

$Transition(\mathtt{f})$ is the set of all transitions between states. $\stackrel{*}{\leadsto}$ is the reflexive transitive closure of $\cup_{e \in \mathcal{E}} \stackrel{e}{\leadsto}$.

Definition 12 *The $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree is a tree defined as follows: (i) nodes are states, (ii) the root is the state $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$, (iii) for each state η_1, the children of η_1 are the states $\{\eta_2 \mid \eta_1 \stackrel{e}{\leadsto} \eta_2 \in Transition(\mathtt{f})\}$.*

Example 13 *Take the `lcs` program whose rules are written in Example 6. The $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba} \rangle$-call tree is displayed on Figure 2.*

Proposition 14 *Assume that $\eta_1 = \langle \mathtt{h}, u_1, \cdots, u_p \rangle \stackrel{*}{\leadsto} \eta_2 = \langle \mathtt{g}, s_1, \cdots, s_m \rangle$. Then we have $(\!|\mathtt{g}(s_1, \cdots, s_m)|\!) \leq (\!|\mathtt{h}(u_1, \cdots, u_p)|\!)$ and thus $(\!|s_i|\!) \leq (\!|\mathtt{h}(u_1, \cdots, u_p)|\!)$ for all $1 \leq i \leq m$.*

PROOF. By virtue of the quasi-interpretation definition. Both the subterm property and the weak monotonicity property are necessary. □

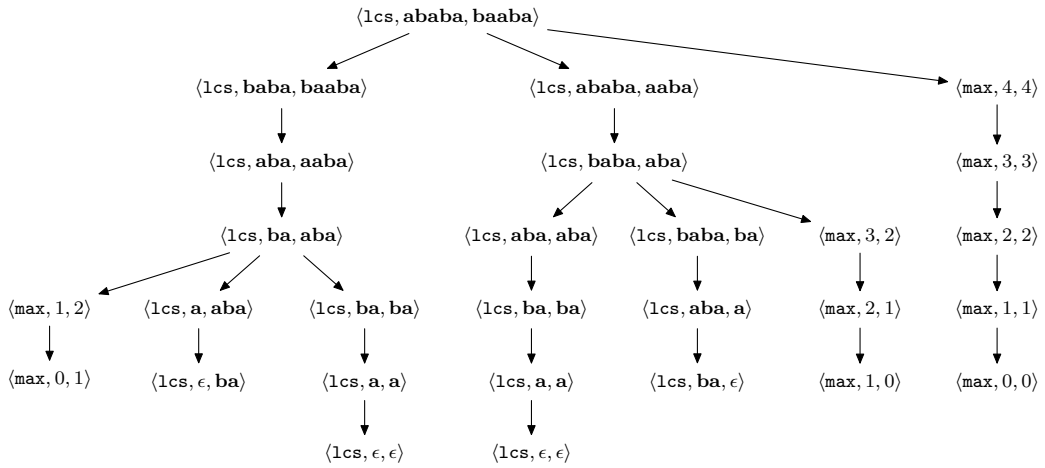

Fig. 2. The $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba} \rangle$-call tree.

The size of a state $\langle \mathsf{g}, s_1, \cdots, s_m \rangle$ is $\sum_{i=1}^{m} |s_i|$.

Lemma 15 *The size of each state of the $\langle \mathsf{f}, t_1, \cdots, t_n \rangle$-call tree is bounded by $d \times (\!|\mathsf{f}(t_1, \cdots, t_n)|\!)$ where d is the maximal arity of a function symbol.*

PROOF. Suppose that $\langle \mathsf{g}, s_1, \cdots, s_m \rangle$ is a state of the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree. It follows from Proposition 14 that $(\!|s_i|\!) \leq (\!|\mathtt{f}(t_1, \cdots, t_n)|\!)$. As s_i is a constructor term, Proposition 10 entails that $|s_i| \leq (\!|s_i|\!)$. Therefore

$$|\langle \mathsf{g}, s_1, \cdots, s_m \rangle| \leq \sum_i |s_i| \leq d \times (\!|\mathtt{f}(t_1, \cdots, t_n)|\!)$$

□

3.4 Upper bound on the complexity

It turns out that we can now state a quite important practical point. Indeed, consider an additive program. By combining Lemma 15 and Proposition 9, we deduce that the size of each state of the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree is bounded by a polynomial in the size of the inputs. So, the size of each intermediate value pushed on a stack is polynomially bounded. This allows to control the output size of functions, even if the computation does not terminate. This is now formalized in the following Theorem.

Theorem 16

- *For an additive program, the size of each state of the $\langle \mathsf{f}, t_1, \cdots, t_n \rangle$-call tree is bounded by $P(m)$ where m is the size of the inputs and P some polynomial.*
- *For an affine program, the size of each state of the $\langle \mathsf{f}, t_1, \cdots, t_n \rangle$-call tree is bounded by $2^{O(m)}$ where m is the size of the inputs.*

- *For an additive program, the size of each state of the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree is bounded by $2^{2^{O(m)}}$ where n is the size of the inputs.*

PROOF. It is a consequence of Lemma 15 and Proposition 9. □

From this result, we can see that the halting problem on a given input is decidable, thus leading to a potential runtime detection of non-termination.

Theorem 17 *Let $\mathtt{f}$ be an additive program and $t_1, \cdots, t_n$ be constructors terms, there is an evaluation procedure which computes the value v if $\mathtt{f}(termone_1, \cdots, termone_n) \downarrow v$ and otherwise returns $\perp$ in exponential time, i.e. in $2^{P(\max_{i=1}^{n} |t_i|)}$, where P is a polynomial which depends on the quasi-interpretation $(\!|-|\!)$.*

PROOF. The proof is based on Cook's simulation [14] of push-down automata with bounded memory. There are similar proofs by Jones [22], Marion-Moyen [32] and by Amadio [1].

We give a sketch of it. Take a program $\mathtt{f}$ and inputs $t_1, \cdots, t_n$. In order to evaluate $\mathtt{f}(t_1, \cdots, t_n)$, we have to compute and memorize the value of each distinct state of the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree. The procedure transforms the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree into a directed acyclic graph which is the essence of memoization techniques used in mentioned works above. Following Lemma 15, the size of states is bounded by $O((\!|\mathtt{f}(t_1, \cdots, t_n)|\!))$. And so there are at most $2^{O((\!|\mathtt{f}(t_1, \cdots, t_n)|\!))}$ different states. Now, suppose that $\mathtt{f}$ is additive. It follows that the number of states is bounded by $2^{P(\max_{i=1}^{n} |t_i|)}$. So, the computation is exponential. □

Corollary 18 *Assume that $\mathtt{f}$ is an affine (resp. multiplicative) program. Given the inputs $t_1, \cdots, t_n$, there is an evaluation procedure which computes the value v if $t \downarrow v$ and otherwise returns $\perp$ in double exponential time, i.e. in $2^{2^{P(\max_{i=1}^{n} |t_i|)}}$ (resp. in triple exponential time, i.e. in $2^{2^{2^{P(\max_{i=1}^{n} |t_i|)}}}$), where P is a polynomial which depends on the quasi-interpretation $(\!|-|\!)$.*

3.5 Uniform Termination is undecidable

Quasi-interpretations do not ensure termination. Indeed, the rule $\mathtt{f}(x) \to \mathtt{f}(x)$ admits the quasi-interpretation $(\!|\mathtt{f}|\!)(X) = X$ but does not terminate. Moreover, quasi-interpretations do not give enough information to decide uniform termination as stated in the following theorem.

Theorem 19 *It is undecidable to know whether a program which admits a polynomial quasi-interpretation, terminates or not on all inputs.*

PROOF. Senizergues proved in [36] that the uniform termination of non-increasing semi-Thue systems is undecidable. These semi-Thue systems are a particular case of rewriting systems with a quasi-interpretation (simply take the identity polynomial for the unary symbols and 1 for the unique constant ϵ) . The theorem follows immediately. □

3.6 Interpretations

Before proceeding further, there is a certain interest in discussing about termination interpretation proofs and in seeing how they differ from quasi-interpretations.

Definition 20 *A polynomial interpretation is an assignment such that for each* $b \in \mathcal{F} \bigcup \mathcal{C}$

(i) $(\!|b|\!)$ *is a polynomial,*
(ii) $(\!|b|\!)(X_1, \cdots, X_n) > X_i$*, for each variable* X_i *(this condition guarantees strict monotonicity and ensures that the interpretation induces a simplification ordering),*
(iii) for each rule $l \rightarrow r$ *and for each substitution* σ*,* $(\!|l\sigma|\!) > (\!|r\sigma|\!)$*.*

Programs admitting an interpretation terminate. This sort of termination proof was introduced by Lankford [25] and turns out to be a useful tool (see [10,17,13] among others).

Remark 21 *An interpretation is also a quasi-interpretation but the converse is not true.*

Example 22

$$f(\mathbf{0}, y) \rightarrow \mathbf{S}'(y)$$
$$f(\mathbf{S}(x), y) \rightarrow f(x, f(x, y))$$

The above program computes the function defined by $[\![f]\!](n,m) = 2^n + m$. It admits the following interpretation which is affine

$$\begin{aligned}(\!|\mathbf{0}|\!) &= 1\\ (\!|\mathbf{S}|\!)(X) &= 2X+1\\ (\!|\mathbf{S}'|\!)(X) &= X+1\\ (\!|f|\!)(X,Y) &= X+Y+1\end{aligned}$$

The kind of an interpretation is determined according to the interpretation of constructors, in the same way as for quasi-interpretations.

Theorem 23 (Bonfante, Cichon, Marion and Touzet [6])

(1) The set of functions computed by programs admitting additive *interpretations is exactly the set of functions computable in* polynomial time*.*
(2) The set of functions computed by programs admitting affine *interpretations is exactly the set of functions computable in* linear exponential time*, that is in time bounded by* $2^{O(n)}$*.*
(3) The set of functions computed by programs admitting multiplicative *interpretations is exactly the set of functions computable in* linear doubly exponential time*, that is in time bounded by* $2^{2^{O(n)}}$*.*

4 Synthesis of Quasi-interpretations

One virtue of assignments over reals is that it affords a procedure to determine a program quasi-interpretation. Of course, such procedure cannot be achieved over natural numbers (or rational numbers) because Matiasevitch's [33] demonstrated that Hilbert's tenth problem is undecidable.

In this section, we shall see that the finding quasi-interpretations over reals is solvable because it is a consequence of Tarski's Theorem [37]. For this, we shall consider two distinct problems. The first one, the *verification problem*, is as follows.

inputs: A program `f` and an assignment $(\!|-|\!)$.
problem: Is $(\!|-|\!)$ a quasi-interpretation for `f` ?

The second one, the *synthesis problem*, is as follows

inputs: A program `f`.
problem: Does there exists an assignment $(\!|-|\!)$ which is a quasi-interpretation for `f` ?

Both problems are solvable when we restrict the class of assignment to the class of **Max-Poly** functions.

Definition 24 *The class of* ***Max-Poly*** *functions contains constant functions ranging over non negative rational numbers and is closed by projections,* max*, addition, multiplication and composition.*

Before proceeding to the main discussion, it is convenient to have a normal representation of function in **Max-Poly**.

Proposition 25 (Normalization) *A* ***Max-Poly*** *function Q is defined thus*

$$Q(X_1, \ldots, X_n) = \max(P_1(X_1, \ldots, X_n), \ldots, P_k(X_1, \ldots, X_n))$$

where P_i is a polynomial.

PROOF. This is due to the fact that max is distributive with $+$ and $\times$ over the non-negative reals. □

We say that the max-degree of Q is k and the degree of Q is the maximum degree of the polynomials $P_1, \ldots, P_k$.

The quasi interpretations of all examples belong to the class **Max-Poly**. Actually, it appears that the class of **Max-Poly** quasi-interpretations is sufficient for daily programs.

Now consider a **Max-Poly** assignment $(\!|-|\!)$ of a program `f`. Take a rule $l \to r$ and define

$$S_{l\to r} = \forall X_1, \ldots X_p \geq 0 : \bigvee_{i=1..n} \bigwedge_{j=1..k} P_i(X_1, \ldots, X_p) \geq Q_j(X_1, \ldots, X_p)$$

where $(\!|l|\!) = \max(P_1, \ldots, P_n)$, $(\!|r|\!) = \max(Q_1, \ldots, Q_k)$ and $X_1, \ldots, X_p$ are all the variables of $(\!|l|\!)$. (Recall that the variables of $(\!|r|\!)$ are also variables of $(\!|l|\!)$.)

We see that the first order formula $S_{l\to r}$ is true iff $(\!|l|\!) \geq (\!|r|\!)$.

Theorem 26 *The verification problem for* ***Max-Poly*** *assignments is decidable in double exponential time with respect to the maximum arity of a symbol.*

PROOF. Tarski showed that the first-order theory for reals containing the addition $+$, the multiplication $\times$, the equality $=$, the order $>$ with variables over reals and rational constant is decidable [37]. In order to solve the verification problem, we have to decide whether or not the following first order

formula is true.

$$S_{\mathcal{E}} = \bigwedge_{l \to r \in \mathcal{E}} S_{l \to r}$$

This is performed by Tarski's decision procedure. Collins [12] established that such procedure is at most doubly exponential in the number of quantifiers. In our case, it corresponds to the maximum arity of symbols. □

Theorem 27 *The synthesis problem for* ***Max-Poly*** *assignment of bounded degree and bounded* max*-degree is decidable in double exponential time in the size of the program.*

PROOF. Without loss of generality, we restrict ourselves to unary functions. Functions with many variables are handled in the same way but with more coefficients and indexes. By Theorem hypothesis, we assume that the degree is d and the max-degree is k.

Suppose that there are n symbols, constructors or functions, $b_1, \ldots, b_n$. The assignment of b_i is

$$(\!|b_i|\!)(X) = \max(P_1^{b_i}(X), \ldots, P_k^{b_i}(X)) \qquad \text{where } P_m^{b_i} = \sum_{j=0}^{d} a_{b_i,m,j} X^j$$

Now, we have to guess polynomial coefficients by proving the validity of the formula:

$$\exists a_{b_1,1,0} \ldots a_{b_1,k,d}, \ldots, a_{b_n,1,0}, \ldots, a_{b_n,k,d} : S_{\mathcal{E}}$$

Lastly, we need to verify that the subterm and the weak monotonicity properties and the fact that the coefficient of degree 0 for constructors is ≥ 1.

The total number of quantifiers is $k \times (d+1) \times n$. So, the decision procedure is doubly exponential in the size of the program. □

Remark 28 *In practice, each program appears to admit a* ***Max-Poly*** *quasi-interpretation with low degrees, usually no more than* 2 *for both the degree of polynomials and the arity of* max*.*

Although a solution of the decision of **Max-Poly** synthesis problem is presented above, yet the procedure for carrying out the decision is complex. There is need of specific methods for finding quasi-interpretations which are in a smaller class but which are relevant. For this reason, Amadio [1] considered

the max-plus algebra over rational numbers. A program which admits a quasi-interpretation over the max-plus algebra are related to non-size increasing according to Hofmann [19]. Amadio established that the synthesis of max-plus quasi-interpretation is in NPTIME-hard and NPTIME-complete in the case of multi-linear assignments.

5 Termination

We now focus on termination which plays the role of a mold capturing certain algorithm patterns. We obtain a finer control resource by the combination of termination tools and quasi-interpretation. Here, we consider Recursive Path Orderings which are simplification orderings and so well-founded. Among the pioneers of this subject, there are Plaisted [34], Dershowitz [16], Kamin and Lévy [23]. Finally, Krishnamoorthy and Narendran in [24] have proved that deciding wheater a program terminates by Recursive Path Orderings is a NP-complete problem.

5.1 Extension of an ordering to sequences

Suppose that $\preceq$ is a partial ordering and $\prec$ its strict part. We describe two extensions of $\prec$ to sequences of the same length.

Definition 29 *The product extension*[1] *of $\prec$ over sequences, noted $\prec^p$, is defined as follows.*
We have $(m_1, \cdots, m_k) \prec^p (n_1, \cdots, n_k)$ if and only if (i) $\forall i \leq p, m_i \preceq n_i$ and (ii) $\exists j \leq k$ such that $m_j \prec n_j$.

Definition 30 *The lexicographic extension of $\prec$, noted $\prec^l$, is defined as follows.*
We have $(m_1, \cdots, m_k) \prec^l (n_1, \cdots, n_k)$ if and only if there exists an index j such that (i) $\forall i < j, m_i \preceq n_i$ and (ii) $m_j \prec n_j$.

The product extension is a restriction of the usual multi-set extension. Notice also that two sequences ordered by the product extension are ordered lexicographically.

[1] Unlike [32], we have decided to present the product extension instead of the permutation extension. This simplifies the presentation without loss of generality. Actually, there is a tedious procedure to transform the rules in order to prove termination by product ordering.

$$\frac{(s_1,\cdots,s_n) \prec^{p}_{rpo} (t_1,\cdots,t_n)}{\mathbf{c}(s_1,\cdots,s_n) \prec_{rpo} \mathbf{c}(t_1,\cdots,t_n)}\ \mathbf{c}\in\mathcal{C} \qquad \frac{s = t_i \text{ or } s \prec_{rpo} t_i}{s \prec_{rpo} \mathtt{f}(\ldots,t_i,\ldots)}\ \mathtt{f}\in\mathcal{F}\bigcup\mathcal{C}$$

$$\frac{\forall i\ s_i \prec_{rpo} \mathtt{f}(t_1,\cdots,t_n)}{\mathbf{c}(s_1,\cdots,s_m) \prec_{rpo} \mathtt{f}(t_1,\cdots,t_n)}\ \mathtt{f}\in\mathcal{F},\mathbf{c}\in\mathcal{C}$$

$$\frac{\forall i\ s_i \prec_{rpo} \mathtt{f}(t_1,\cdots,t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{g(s_1,\cdots,s_m) \prec_{rpo} \mathtt{f}(t_1,\cdots,t_n)}\ \mathtt{f},\mathtt{g}\in\mathcal{F}$$

$$\frac{(s_1,\cdots,s_n) \prec^{st(\mathtt{f})}_{rpo} (t_1,\cdots,t_n) \qquad \mathtt{f}\approx_{\mathcal{F}}\mathtt{g} \qquad \forall i\ s_i \prec_{rpo} \mathtt{f}(t_1,\cdots,t_n)}{\mathtt{g}(s_1,\cdots,s_n) \prec_{rpo} \mathtt{f}(t_1,\cdots,t_n)}\ \mathtt{f},\mathtt{g}\in\mathcal{F}$$

Fig. 3. Definition of $\prec_{rpo}$

5.2 Recursive path ordering with status

Let $\prec_{\mathcal{F}}$ be an ordering on $\mathcal{F}$ and $\approx_{\mathcal{F}}$ be a compatible equivalence relation such that if $\mathtt{f}\approx_{\mathcal{F}}\mathtt{g}$ then $\mathtt{f}$ and $\mathtt{g}$ have the same arity. The quasi-ordering $\preceq_{\mathcal{F}}=\prec_{\mathcal{F}}\cup\approx_{\mathcal{F}}$ is a precedence over $\mathcal{F}$.

Definition 31 *A status st is a mapping which associates to each function symbol $\mathtt{f}$ of $\mathcal{F}$ a status $st(\mathtt{f})$ in $\{p,l\}$. A status is compatible with a precedence $\preceq_{\mathcal{F}}$ if it satisfies the fact that if $\mathtt{f}\approx_{\mathcal{F}}\mathtt{g}$ then $st(\mathtt{f})=st(\mathtt{g})$.*

Throughout, we assume that status are compatible with precedences.

Definition 32 *Given a precedence $\preceq_{\mathcal{F}}$ and a status st, the recursive path ordering $\prec_{rpo}$ is defined in Figure 3.*

When $st(\mathtt{f}) = p$, the status of $\mathtt{f}$ is said to be product. In that case, the arguments are compared with the product extension of $\prec_{rpo}$. Otherwise, the status is said to be lexicographic.

A rule $l\to r$ is decreasing if we have $r\prec_{rpo} l$. A program is ordered by $\prec_{rpo}$ if there is a precedence on $\mathcal{F}$ and a status st such that each rule is decreasing.

Theorem 33 (Dershowitz [16]) *Each program which is ordered by $\prec_{rpo}$ terminates on all inputs.*

Remark 34 *The definition of $\prec_{rpo}$ takes into account the difference between function symbols and constructors. Actually, a precedence $\preceq_{\mathcal{F}}$ could be extended canonically over $\mathcal{C}\cup\mathcal{F}$ by saying that (i) constructors are smaller than*

function symbols, (ii) two constructors are incomparable, (iii) constructors have a product status.

Example 35

(1) The shuffle program rearranges two words. It terminates with a product status.

$$\begin{aligned} \mathrm{shuffle}(\boldsymbol{\epsilon}, y) &\to y \\ \mathrm{shuffle}(x, \boldsymbol{\epsilon}) &\to x \\ \mathrm{shuffle}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathbf{i}(\mathbf{j}(\mathrm{shuffle}(x, y))) \qquad \mathbf{i}, \mathbf{j} \in \{\mathbf{0}, \mathbf{1}\} \end{aligned}$$

(2) The following program reverses a word by tail-recursion. It terminates with a lexicographic status.

$$\begin{aligned} \mathrm{reverse}(\boldsymbol{\epsilon}, y) &\to y \\ \mathrm{reverse}(\mathbf{i}(x), y) &\to \mathrm{reverse}(x, \mathbf{i}(y)) \qquad \mathbf{i} \in \{\mathbf{0}, \mathbf{1}\} \end{aligned}$$

(3) The program `sort` of Example 5 terminates by setting `if then else` $\prec$ `insert` $\prec$ `sort`. Each function symbol has a product status.
(4) The `lcs` program describes of Example 6 is ordered by taking `max` $\prec$ `lcs`, and both symbols have a product status.

5.3 Extensional Characterization

The orderings considered are special cases of the more general ones and in particular of *Multiset Path Ordering* and *Lexicographic Path Ordering*. Nevertheless, they characterize the same set of functions. Say that a $\mathrm{RPO}_{\mathrm{Pro}}$-program is a program in which each function symbol has a product status. Following the result of Hofbauer [18], we have

Theorem 36 *The set of functions computed by RPO_{Pro}-programs is exactly the set of primitive recursive functions.*

Now, say that a $\mathrm{RPO}_{\mathrm{Lex}}$-program is a program in which each function symbol has a lexicographic status. Weiermann [38] has established that

Theorem 37 *The set of functions computed by RPO_{Lex}-programs is exactly the set of multiply-recursive functions.*

5.4 Consequences of termination proofs

We write $s \trianglelefteq t$ to say that s is a subterm of t.

Proposition 38

(1) For each constructor term t and s, $s \prec_{rpo} t$ iff $s \trianglelefteq t$.
(2) For each constructor term $s_1, \cdots, s_n$ and $t_1, \cdots, t_n$, $(s_1, \cdots, s_n) \prec^x_{rpo} (t_1, \cdots, t_n)$ implies $(s_1, \cdots, s_n) \trianglelefteq^x (t_1, \cdots, t_n)$, where x is a status p or l and $\trianglelefteq^x$ is the corresponding extension based on the subterm relation.
(3) For each constructor term $s_1, \cdots, s_n$ and $t_1, \cdots, t_n$, $(s_1, \cdots, s_n) \prec^x_{rpo} (t_1, \cdots, t_n)$ implies $(|s_1|, \cdots, |s_n|) <^x (|t_1|, \cdots, |t_n|)$, where x is a status p or l and $<^x$ is the corresponding extension of the ordering over natural numbers.

PROOF. The proofs go by induction on the size of terms. □

Lemma 39 *Let $\mathtt{f}$ be a program which is ordered by $\prec_{rpo}$, α be the number of function symbols and d be the maximal arity of function symbols. Assume that the size of each state of the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree is strictly bounded by A. Then the following facts hold:*

(1) If $\langle \mathtt{f}, t_1, \cdots, t_n \rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, s_1, \cdots, s_m \rangle$ then (a) $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$ or (b) $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ and $(s_1, \cdots, s_m) \prec^{st(\mathtt{f})}_{rpo} (t_1, \cdots, t_n)$.
(2) If $\langle \mathtt{f}, t_1, \cdots, t_n \rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, s_1, \cdots, s_m \rangle$ and $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ then the number of states between $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$ and $\langle \mathtt{g}, s_1, \cdots, s_m \rangle$ is bounded by A^d.
(3) The length of each branch of the call-tree is bounded by $\alpha \times A^d$.

PROOF.

(1) Because the rules of the program decrease by $\prec_{rpo}$ and $\overset{-}{\rightsquigarrow}$ is compatible with the rewriting relation.
(2) Let $\langle \mathtt{h}, u_1, \cdots, u_p \rangle$ be state which, in the call-tree, is between $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$ and $\langle \mathtt{g}, s_1, \cdots, s_m \rangle$. Due to the first point of this lemma, we have $\mathtt{h} \approx_{\mathcal{F}} \mathtt{f}$ and $(u_1, \cdots, u_p) \prec^{st\mathtt{f}}_{rpo} (t_1, \cdots, t_n)$. So, by proposition 38(3), we have $(|u_1|, (|u_1|, \cdots, |u_p|) <^{st(\mathtt{f})} (|t_1|, \cdots, |t_n|)$. Since the size of each component is bounded by A and $n \leq d$, the length of the decreasing chain is bounded by A^d.
(3) In each branch, the previous point of the Lemma claims that there are at most A^d states whose function symbols have the same precedence. Next, there are A^d states whose function symbol have the precedence immediately below, and so on. As there are only α function symbols, the length of the branch is bounded by $\alpha \times A^d$.

□

6 Characterizing space bounded computation

6.1 Polynomial space computation

Definition 40 *A RPO^{QI}-program is a program that (i) admits a quasi-interpretation and (ii) which terminates by $\prec_{rpo}$.*

Theorem 41 *The set of functions computed by* additive *RPO^{QI}-programs is exactly the set of functions computable in* polynomial space.

The upper-bound on space-usage is established by Theorem 44. The completeness of this characterization is established by Theorem 65.

Example 42

The Quantified Boolean Formula (QBF) problem is PSPACE complete. It consists in determining the validity of a boolean formula with quantifiers over propositional variables. Without loss of generality, we restrict formulae to $\neg, \vee, \exists$. QBF problem is solved by the following program.

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

$$\mathtt{in}(x, \mathbf{nil}) \to \mathbf{ff} \qquad \mathtt{in}(x, \mathbf{cons}(a, l)) \to \mathtt{or}(x = a, \mathtt{in}(x, l))$$

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

Booleans are encoded by $\{\mathbf{tt}, \mathbf{ff}\}$, variables are encoded by unary integers which are generated by $\{\mathbf{0}, \mathbf{S}\}$. Formulae are built from $\{\mathbf{Var}, \mathbf{Not}, \mathbf{Or}, \mathbf{Exists}\}$. All these symbols are constructors. The main function symbol is `qbf`.

Rules are ordered by $\prec_{rpo}$ by putting

$$\{\mathtt{not}, \mathtt{or}, \mathtt{\_=\_}\} \prec_{\mathcal{F}} \mathtt{in} \prec_{\mathcal{F}} \mathtt{ver} \prec_{\mathcal{F}} \mathtt{qbf}$$

and each function symbol has a product status except `ver` which has a lexicographic status.

They admit the following additive quasi-interpretations :

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

6.2 RPO^{QI}-programs are PSPACE *computable*

We are now establishing that a RPO^{QI}-program $\mathtt{f}$ is computable in polynomial space.

Lemma 43 *Let $\mathtt{f}$ be a RPO^{QI}-program. For each constructor term $t_1, \cdots, t_n$, the space used by a call by value (innermost) interpreter to compute $\mathtt{f}(t_1, \cdots, t_n)$ is bounded by a polynomial in $(\!|\mathtt{f}(t_1, \cdots, t_n)|\!)$.*

PROOF. Take an innermost call by value interpreter, like the one of Figure 1. It builds recursively in a depth first manner the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree, evaluates nodes and backtracks. Put $A = (\!|\mathtt{f}(t_1, \cdots, t_n)|\!)$. The interpreter only needs to store states along a branch of the call-tree. Each state as well as the intermediate results are bounded by $O(A)$. The maximal length of a branch is bounded by $\alpha \times A^d$ by Lemma 39(3). The number of states and results to memorize for the depth first search is bounded by $\alpha \times A^{d+1} \times \beta$ where β is the maximal size of a rule. In other words, β is an upper bound on the width of the call-tree. Therefore, the space used by the interpreter is bounded by $O(A^{d+1})$. $\square$

Theorem 44 *Let $\mathtt{f}$ be an additive RPO^{QI}-program. For each constructor term $t_1, \cdots, t_n$, the space used by a call by value interpreter to compute $\mathtt{f}(t_1, \cdots, t_n)$ is bounded by a polynomial in $\max_{i=1}^{n} |t_i|$.*

PROOF. By Proposition 9, we have $(\!|t_i|\!) \leq O(|t_i|)$. Because quasi-interpretations are polynomially bounded, we have $(\!|\mathtt{f}(t_1, \cdots, t_n)|\!) \leq P(\max_{i=1}^{n} |t_i|)$, for some polynomial P. So the space is bounded by $O(P(\max_{i=1}^{n} |t_i|)^{d+1})$ following Lemma 43. $\square$

6.3 Beyond polynomial space

The kind of quasi-interpretations of constructors gives an upper bound on the space required to evaluate a program.

Theorem 45

- *The set of functions computed by* affine RPO^{QI}*-programs is exactly the set of functions computable in* linear exponential space*, that is in space bounded by* $2^{O(n)}$ *where* n *is the size of the inputs.*
- *The set of functions computed by* multiplicative RPO^{QI}*-programs is exactly the set of functions computable in* linear double exponential space*, that is in space bounded by* $2^{2^{O(n)}}$ *where* n *is the size of the inputs..*

Proofs are very similar to the one of Theorem 44. The kind of quasi-interpretation gives the different upper-bounds on the space-usage as established in Proposition 9.

Completeness of the characterisations is still a consequence of Theorem 65.

7 Characterizing time bounded computation

7.1 Polynomial time computation

Definition 46 *A function symbol* $\mathtt{f}$ *is linear in a program terminating by* $\prec_{rpo}$ *if for each rule* $\mathtt{f}(p_1, \cdots, p_n) \to r$*, then there is at most one occurrence in* r *of a function symbol* $\mathtt{g}$ *with the same precedence than* $\mathtt{f}$*, that is* $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$*.*

Definition 47

(1) A RPO^{QI}_{Pro}*-program is a program that (i) admits a quasi-interpretation, (ii) which terminates by* $\prec_{rpo}$ *and (iii) each function symbol has a product status.*
(2) A RPO^{QI}_{Lin}*-program is a program that (i) admits a quasi-interpretation, (ii) which terminates by* $\prec_{rpo}$*, and (iii) each function symbol is linear and has a lexicographic status.*
(3) A $RPO^{QI}_{Pro+Lin}$*-program is a program that (i) admits a quasi-interpretation, (ii) which terminates by* $\prec_{rpo}$ *and (iii) each function symbol which has a lexicographic status is linear.*

Tail recursive programs are $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-programs as it is illustrated by the reverse program in Example 35. On the other hand, the program that solves QBF in Example 42, is not a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program, because of the definition of `ver` (in the case of $\mathbf{Exists}(n, \phi)$) which leads to two recursive calls with substitution of parameters. Note that lexicographic ordering captures the template of recursion with parameter substitutions which was the key ingredient of the characterization of polynomial space functions by [28] by tiering discipline.

Theorem 48 *The set of functions computed by* additive RPO^{QI}_{Lin}*-programs (resp.* RPO^{QI}_{Pro}*-programs and* $RPO^{QI}_{Pro+Lin}$*-programs) is exactly the set of functions computable in* polynomial time*.*

The upper-bound on time-usage is established by Theorem 53 below. The completeness of this characterization is established by Theorem 64.

Example 49 *The* `lcs` *example 6 is quite interesting and is an illustration of an important observation. Indeed, if one applies the rules of the program following a call by value strategy, one gets an exponentially long derivation chain. But the theorem states that the* `lcs` *function is computable in polynomial time. Actually, one should be careful not to confuse the algorithm and the function it computes. This function (length of the longest common subsequence) is a classical textbook example of so called "dynamic programming" (see chapter 16 of [15]) and can in this way be computed in polynomial time.*

So, the theorem does not characterize the complexity of the algorithm, which we should call its explicit *complexity but the complexity of the function computed by this algorithm, which we should dub its* implicit *complexity.*

7.2 $RPO^{QI}_{Pro+Lin}$-programs are PTIME *computable*

In order to avoid an exponential explosion like, for instance, in the `lcs` case, we switch from the call-by-value semantics previously defined to a call-by-value semantics with cache, see Figure 4. Hence, we simulate dynamic programming techniques, which consist in storing each result of a function call in a table and avoiding to recompute the same function call if it is already in the table. This technique is inspired from Ben-Amran and Jones' rereading ([3]) of Cook simulation technique over 2 way push-down automata ([14]) and is called memoization.

The expression $\langle C, t\rangle \Downarrow \langle C', v\rangle$ means that the computation of t is v given a program `f` and an initial cache C. The final cache C' contains C and each call which has been necessary to complete the computation.

More precisely, say that a configuration is a list such as $(\mathtt{g}, v_1, \cdots, v_m, v)$ where $\langle \mathtt{g}, v_1, \cdots, v_m\rangle$ is a state, and $[\![\mathtt{g}]\!](v_1, \cdots, v_m) = v$. When a term $\mathtt{g}(v_1, \cdots, v_m)$ is considered, we search for a configuration $(\mathtt{g}, v_1, \cdots, v_m, v)$ in the current cache C. If such configuration exists, we use it to short-cut the computation and so we return v. Otherwise, we apply a program equation, say $l \to r$, by matching $\mathtt{g}(v_1, \cdots, v_m)$ with l. Then, we update C by adding the configuration $(\mathtt{g}, v_1, \cdots, v_m, v)$ to the current cache C.

Figure 5 shows what happens to the $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba}\rangle$-call tree when mem-

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

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad (\mathtt{f}, v_1, \cdots, v_n, v) \in C_n}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow \langle C_n, v\rangle}\ (Read)$$

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C_n, r\sigma\rangle \Downarrow \langle C, v\rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow \langle C \cup (\mathtt{f}, v_1, \cdots, v_n, v), v\rangle}$$

$$(Update)$$

Fig. 4. Call-by-value interpreter with Cache of $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}\rangle$.

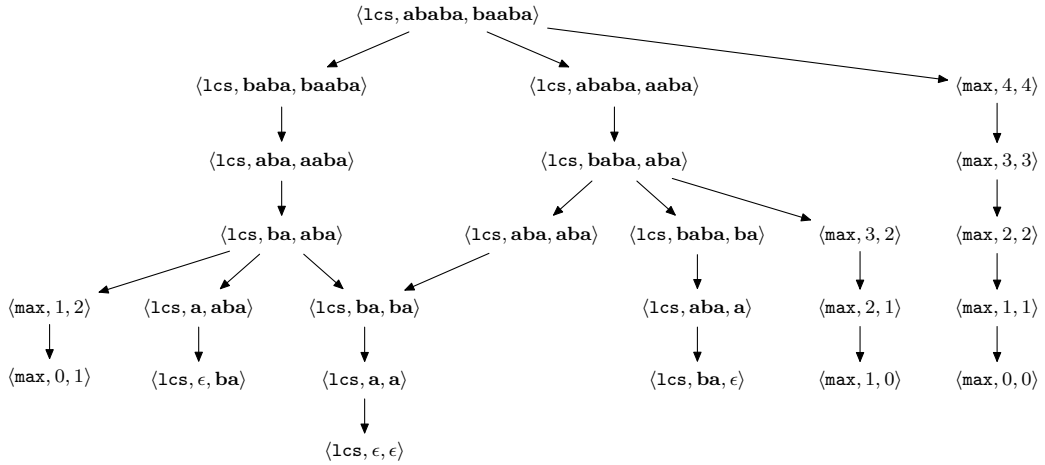

Fig. 5. The $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba}\rangle$-call tree with memoization.

oization is applied. Notice that identical subtrees are merged and the call-tree becomes a directed acyclic graph.

The key point for additive programs, is to establish that the size of a cache C is polynomially bounded in the size of the input arguments.

Lemma 50 *Suppose that* $\langle C_0, \mathbf{f}(t_1, \cdots, t_n)\rangle \Downarrow \langle C, v\rangle$. *The size of the final cache* C *is bounded by a polynomial in* $(\!|\mathbf{f}(t_1, \cdots, t_n)|\!)$.

PROOF. Define $C_{\mathtt{g}}$ as the set of m-uplets of $\mathcal{T}(\mathcal{C})$-terms which are the arguments of states of $\mathtt{g}$. That is, $(u_1, \cdots, u_m) \in C_{\mathtt{g}}$ iff $(\mathtt{g}, u_1, \cdots, u_m, v) \in C$. We have

$$\#C = \sum_{\mathtt{g} \in \mathcal{F}} \#C_{\mathtt{g}} \tag{1}$$

where we write $\#S$ for the cardinal of a set S.

To give an upper-bound on the cardinality of $C_{\mathtt{g}}$, we define two sets $C^{\vee}_{\mathtt{g}}$ and $C^{\wedge}_{\mathtt{g}}$. The idea is to separate the calls which come from functions of strictly higher precedence and the ones which come from functions of the same precedence. Consider the $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$-call tree. Say that the covering graph of $\mathtt{g}$ is the subgraph of the $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$-call tree obtained by removing all states which are not labeled by functions $\mathtt{h} \approx_{\mathcal{F}} \mathtt{g}$. Define two sets $C^{\vee}_{\mathtt{g}}$ and $C^{\wedge}_{\mathtt{g}}$ as follows. $C^{\vee}_{\mathtt{g}}$ contains all the roots of the covering graph of $\mathtt{g}$ labeled by $\mathtt{g}$, and $C^{\wedge}_{\mathtt{g}}$ contains all the other nodes of the covering graph labeled by $\mathtt{g}$.

- We consider the $C^{\vee}_{\mathtt{g}}$'s. We have $\#C^{\vee}_{\mathtt{f}} = 1$. Suppose that $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, but $\mathtt{f} \neq \mathtt{g}$. By definition, we have $\#C^{\vee}_{\mathtt{g}} = 0$.
  Suppose that $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. Then, $(u_1, \cdots, u_m) \in C^{\vee}_{\mathtt{g}}$. It follows that the cardinality of $C^{\vee}_{\mathtt{g}}$ is bounded by

$$\#C^{\vee}_{\mathtt{g}} \leq \sum_{\mathtt{g} \prec_{\mathcal{F}} \mathtt{h}} \#C_{\mathtt{f}} \tag{2}$$

- We consider $C^{\wedge}_{\mathtt{g}}$.

(1) The status of $\mathtt{g}$ is product. Proposition 38 states that sub-calls of the same rank starting from $\mathtt{f}(v_1, \cdots, v_n)$ have arguments which are subterms of the v_i's. Therefore there are at most $\prod_{i\leq n}(|v_i|+1)$ such sub-calls. It follows from Lemma 15 that the number of sub-calls is bounded

$$\prod_{i\leq n}(|v_i|+1) \leq (d \times (\!|\mathtt{f}(t_1, \cdots, t_n)|\!))^d \tag{3}$$

where d is the maximal arity of a function symbol.

(2) The status of $\mathtt{g}$ is lexicographic. But by definition of $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-programs, there is at most one recursive call starting from $\mathtt{g}$ for each rule application. Lemma 39(3) entails that the maximal length of a branch is $(\!|\mathtt{f}(t_1, \cdots, t_n)|\!)^d$ which is also a bound on the number of successive calls initiated by $\mathtt{f}$.

From both previous points, we obtain that

$$\#C^{\wedge}_{\mathtt{g}} \leq (\#C^{\vee}_{\mathtt{g}} + 1) \times d^d \times (\!|\mathtt{f}(t_1, \cdots, t_n)|\!)^d \tag{4}$$

Finally, we have

$$\#C_{\mathtt{g}} \leq \#C^{\vee}_{\mathtt{g}} + \#C^{\wedge}_{\mathtt{g}} \tag{5}$$

By combining (1), (2), (4), and (5), we see that the cardinality of C is polynomially bounded in $(\!|\mathtt{f}(t_1, \cdots, t_n)|\!)$. □

Example 51 *Figures 6 and 7 show the* $\mathtt{lcs}$- *and* $\mathtt{max}$-*covering graphs in the* $\langle \mathtt{lcs}, \boldsymbol{ababa}, \boldsymbol{baaba}\rangle$-*call tree. Nodes in* $C^{\vee}_{\mathtt{g}}$ *have been squared while nodes of* $C^{\wedge}_{\mathtt{g}}$ *have been circled* ($\mathtt{g} \in \{\mathtt{lcs}, \mathtt{max}\}$).

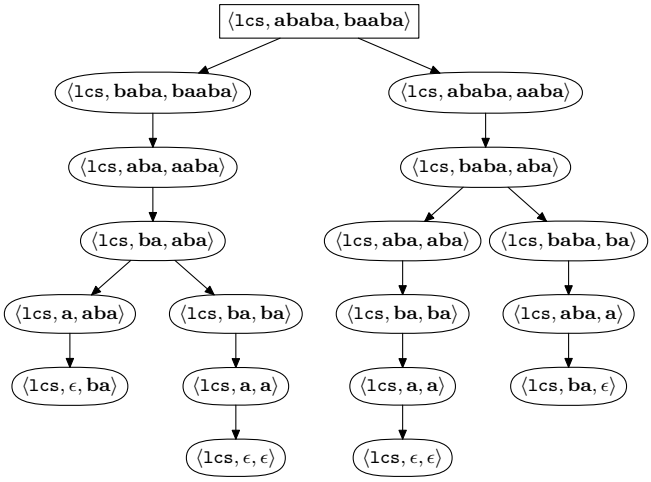

Fig. 6. The `lcs`-covering graphs of the call-tree.

Fig. 7. The `max`-covering graphs of the call-tree.

Lemma 52 *Let $\mathbf{f}$ be a $RPO^{QI}_{Pro+Lin}$-program. For each constructor term $t_1, \cdots, t_n$, the runtime of the call by value interpreter with cache to compute $\mathbf{f}(t_1, \cdots, t_n)$ is bounded by a polynomial in $(\!|\mathbf{f}(t_1, \cdots, t_n)|\!)$.*

PROOF. Since an evaluation procedure memorizes all necessary configurations, the runtime is at most quadratic in the size of the cache. Note that the exact runtime depends on the implementation strategy and in particular on the cache management. □

Theorem 53 *Let $\mathbf{f}$ be an additive $RPO^{QI}_{Pro+Lin}$-program (resp. RPO^{QI}_{Pro}-program and RPO^{QI}_{Lin}-program). For each constructor term $t_1, \cdots, t_n$, the runtime to compute $\mathbf{f}(t_1, \cdots, t_n)$ is bounded by a polynomial in $\max_{i=1}^{n} |t_i|$.*

PROOF. By Proposition 9, we have $(\!|t_i|\!) \leq O(|t_i|)$. So, for some polynomial P we have $(\!|\mathtt{f}(t_1, \cdots, t_n)|\!) \leq P(\max_{i=1}^{n} |t_i|)$, because quasi-interpretations are polynomially bounded. Lemma 52 implies that the time is bounded by a polynomial in $\max_{i=1}^{n} |t_i|$. □

In the general case, memoization is not used because one cannot decide which results will be reused and the cache may become too big to be really useful. In our particular case, the termination ordering gives enough information on the structure of the program to minimize the cache [30].

We see that if a function symbol is linear, see Definition 46, then no result needs to be recorded. More generally, consider the $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$-call tree. When evaluating $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$, the call by value semantics with cache store all values. Say that a separation set N is a set of states such that each chain starting from the root state $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$ meets a state of N. If we know the value of each state of N, then values of the states below N-states are useless in order to determine $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$. And, we can forget them. Therefore, it is sufficient to store in a cache a separation set N for each function symbol. Now say that a separation set N is minimal if for each state $s \in N$, $N \setminus \{s\}$ is not a separation net. We can require an implementation to keep a minimal separation set. To perform dynamically, we have to compare configurations in the cache. Take two configurations $(\mathtt{f}, t_1, \cdots, t_n, t)$ and $(\mathtt{g}, s_1, \cdots, s_m, s)$. If $\mathtt{f}(t_1, \cdots, t_n) \prec_{rpo} \mathtt{g}(s_1, \cdots, s_m)$ then we do not need anymore the configuration $(\mathtt{f}, t_1, \cdots, t_n, t)$ and we can erase it from the cache.

7.3 Beyond polynomial time

Theorem 54

- *The set of functions computed by* affine $RPO^{QI}_{Pro+Lin}$*-programs (resp.* RPO^{QI}_{Pro}*-program and* RPO^{QI}_{Lin}*-program) is exactly the set of functions computable in* linear exponential time*, that is in time bounded by* $2^{O(n)}$.
- *The set of functions computed by* multiplicative $RPO^{QI}_{Pro+Lin}$*-programs (resp.* RPO^{QI}_{Pro}*-program and* RPO^{QI}_{Lin}*-program) is exactly the set of functions computable in* linear double exponential time*, that is in time bounded by* $2^{2^{O(n)}}$.

Proofs are very similar to the one of Theorem 53. The kind of quasi-interpretation gives the different upper-bounds on the time-usage as established in Proposition 9.

Completeness of the characterizations is still a consequence of Theorem 64.

8 Simulation of Parallel Register Machines

8.1 Parallel Register Machines

Following [28], we introduce *Parallel Register Machines (PRM)* which model the essential features of both traditional sequential computing like Turing Machines and alternating computations.

A PRM works over the word algebra $\mathbb{W}$ generated by the constructors $\{\mathbf{0}, \mathbf{1}, \boldsymbol{\epsilon}\}$. In order to have a choice mechanism to simulate alternation by the fork operation, we define an ordering $\blacktriangleleft$ on $\mathbb{W}$: $\boldsymbol{\epsilon} \blacktriangleleft y$, $\mathbf{0}(x) \blacktriangleleft \mathbf{1}(y)$, $\mathbf{i}(x) \blacktriangleleft \mathbf{i}(y)$ if and only if $x \blacktriangleleft y$. We define the operations $\min_{\blacktriangleleft}$ and $\max_{\blacktriangleleft}$ wrt $\blacktriangleleft$.

Definition 55 *A 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) 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 $\Pi \to \mathbb{W}$ which stores register value. We note $\{\pi \leftarrow \pi'\}F$ to mean that the value of the register π is π', the other registers stay unchanged.

Definition 56 *Given M as above we define a semantic partial-function $\boldsymbol{eval}$: $\mathbb{N} \times S \times \mathbb{W}^m \mapsto \mathbb{W}$, that maps the result of the machine in a "time bound" given by the first argument.*

- $\boldsymbol{eval}(0, s, F)$ *is undefined.*
- *If $com(s)$ is $\boldsymbol{Succ}(\pi = \boldsymbol{i}(\pi), s')$ then $\boldsymbol{eval}(t+1, s, F) = \boldsymbol{eval}(t, s', \{\pi \leftarrow \boldsymbol{i}(\pi)\}F)$. 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 $\boldsymbol{eval}(t+1, s, F) = \boldsymbol{eval}(t, s', \{\pi \leftarrow \mathbf{p}(\pi)\}F)$ where $\mathbf{p}$ is the predecessor function on $\mathbb{W}$;*
- *If $com(s)$ is $\boldsymbol{Branch}(\pi, s', s'')$ then $\boldsymbol{eval}(t+1, s, F) = \boldsymbol{eval}(t, r, F)$, 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*
$\boldsymbol{eval}(t+1, s, F) = \min_{\blacktriangleleft}(\boldsymbol{eval}(t, s', F), \boldsymbol{eval}(t, s'', F))$*;*
- *If $com(s)$ is $\boldsymbol{Fork}_{max}(s', s'')$ then*
$\boldsymbol{eval}(t+1, s, F) = \max_{\blacktriangleleft}(\boldsymbol{eval}(t, s', F), \boldsymbol{eval}(t, s'', F))$*;*
- *If $com(s)$ is $\boldsymbol{End}$ then $\boldsymbol{eval}(t+1, s, F) = F(\text{OUTPUT})$.*

Definition 57 *Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A function $\phi : \mathbb{W}^k \to \mathbb{W}$ is PRM-computable in time T if there is a PRM M such that for each $(w_1, \cdots, w_k) \in \mathbb{W}^k$, we have*

$$\texttt{eval}(T(\max_{i=1}^{k} |w_i|), \text{BEGIN}, F_0) = \phi(w_1, \cdots, w_k)$$

where $F_0(\pi_i) = w_i$ for $i = 1..k$ and otherwise $F_0(\pi_j) = \epsilon$.

8.2 Space and Time bounded computation

There are pleasingly well-known tight connection between space used by a Turing machine and time used by PRM. The essence of the translation comes from the works [35,9].

Theorem 58 *A function ϕ is computable in polynomial (resp. exponential, doubly exponential) space iff ϕ is PRM-computable in polynomial time (resp. exponential, doubly exponential).*

A *Register machines (RM)* is a PRM without fork commands. A Turing machine can be simulated linearly in time by a RM.

Proposition 59 *A function ϕ is computable in polynomial (respectively exponential, doubly exponential) time iff ϕ is RM-computable in polynomial time (resp. exponential, doubly exponential).*

8.3 Time bounded simulation

Without loss of generality, we consider only unary function in the following. It would be laborious to specify this simulation in full details otherwise.

Lemma 60 (Lexicographic Plug and play lemma) *Assume that $\phi : \mathbb{W} \to \mathbb{W}$ is a PRM-computable function in time bounded by T. Define f by*

$$\begin{aligned} f : \mathbb{N} \times \mathbb{W} &\to \mathbb{W} \\ (n, w) &\mapsto \phi(w) \quad \text{if } n > T(|w|) \\ (n, w) &\mapsto \perp \qquad \text{otherwise} \end{aligned}$$

Then,

(1) the function f is computed by an additive RPO^{QI}-program,
(2) and, if f is computed by a RM, then f is computable by an additive RPO^{QI}_{Lin}-program.

PROOF. The simulation of the PRM is done by following the rules of `eval` given above. For this, the set of constructors is $\mathcal{C} = \{\mathbf{0}, \mathbf{1}, \mathbf{s}, \diamond, \boldsymbol{\epsilon}\} \cup S$ where S is the set of states.

We show below programs to compute $\min_{\blacktriangleleft}$ and $\max_{\blacktriangleleft}$.

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

with $\mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}$.

Next, we write a program to compute `eval`.

(a) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1 \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')$,

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

(c) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \mathbf{0}(\pi_j), \cdots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \cdots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$,

(d) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \mathbf{1}(\pi_j), \cdots, \pi_m) \to \mathtt{Eval}(t, s'', \pi_1, \cdots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$,

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

(f) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{max}(\mathtt{Eval}(t, s', \pi_1, \cdots, \pi_m), \mathtt{Eval}(t, s'', \pi_1, \cdots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\max}(s', s'')$,

(g) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \pi_m$, if $com(s) = \mathbf{End}$,

(h) $\mathtt{Eval}(\diamond, s, \pi_1, \cdots, \pi_m) \to \bot$.

Finally, put $\mathtt{f}(t, w) \to \mathtt{Eval}(t, \text{BEGIN}, w, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon})$. It is routine to check that $f = [\![\mathtt{f}]\!]$. This program admits the following quasi-interpretations:

$$\begin{array}{ccccc}
(\!|\boldsymbol{\epsilon}|\!) = 1 & & (\!|\diamond|\!) = 0 & & \forall q \in S, (\!|q|\!) = 1 \\
(\!|\mathbf{0}|\!)(X) = X + 1 & & (\!|\mathbf{1}|\!)(X) = X + 1 & & (\!|\mathbf{s}|\!)(X) = X + 1
\end{array}$$

$$
\begin{aligned}
(\!|\texttt{min}|\!)(W, W') &= \max(W, W') \\
(\!|\texttt{max}|\!)(W, W') &= \max(W, W') \\
(\!|\texttt{Eval}|\!)(T, S, \Pi_1, \cdots, \Pi_m) &= T + S + \sum_{i=1}^{m} \Pi_i \\
(\!|\texttt{f}|\!)(T, X) &= T + X + m
\end{aligned}
$$

The status of each function symbol is lexicographic. The precedence satisfies $\{\texttt{min}, \texttt{max}\} \prec_{\mathcal{F}} \texttt{Eval} \prec_{\mathcal{F}} \texttt{f}$. We see that each rule is decreasing by $\prec_{rpo}$. Therefore, `f` is a RPO^{QI}-program. Now, observe that `Eval` has always one occurrence in the right hand side of the rules except in the fork cases. So, `f` is a RPO$^{\text{QI}}_{\text{Lin}}$-program if f is computed by a RM. $\square$

In [32], the simulation of RM is performed by a RPO$^{\text{QI}}_{\text{Pro}}$-program in a different manner because the status of function symbols is product and not lexicographic as in the above result. For this reason, we give details of the simulation of a time bounded function in the case where symbols have a product status.

Lemma 61 (Product Plug and play lemma) *Assume that $\phi : \mathbb{W} \to \mathbb{W}$ is a RM-computable function in time bounded by T. Define f by*

$$
\begin{aligned}
f : \mathbb{N} \times \mathbb{W} &\to \mathbb{W} \\
(n, w) &\mapsto \phi(w) \quad \textit{if } n > T(|w|) \\
(n, w) &\mapsto \bot \qquad \textit{otherwise}
\end{aligned}
$$

Then, f is computable by an additive RPO$^{QI}_{Pro}$-program.

PROOF. Compared with the previous proof, the simulation is performed in bottom-up way. For this, we use an extra constructor $\mathbf{c}$ to encode tuples. And we define `Step` which gives the next configuration.

(a) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \pi_m)) \to \mathbf{c}(s', \pi_1 \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')$

(b) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m)) \to \mathbf{c}(s', \pi_1, \cdots, \pi_j, \cdots, \pi_m)$
if$com(s) = \mathbf{Pred}(\pi_j = \mathbf{p}(\pi_j), s')$

(c) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \mathbf{0}(\pi_j), \cdots, \pi_m) \to \mathbf{c}(s', \pi_1, \cdots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$

(d) $\mathtt{Step}(\mathbf{c}(s, \pi_1, \cdots, \mathbf{1}(\pi_j), \cdots, \pi_m) \to \mathbf{c}(s'', \pi_1, \cdots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$

(e) $\mathtt{Step}(\mathbf{c}(s, \pi_1, \ldots, \pi_m) \to \pi_m$
if $com(s) = \mathbf{End}$

(f) $\mathtt{Step}(\mathbf{i}(x)) \to \mathbf{i}(x), \mathbf{i} \in \{\mathbf{0}, \mathbf{1}\}$

The simulation is made by

$$\mathtt{Eval}(\boldsymbol{\epsilon}, x) \to x \qquad \mathtt{Eval}(\mathbf{s}(t), x) \to \mathtt{Step}(\mathbf{c}(\mathtt{Eval}(t, x)))$$

The rules are ordered by putting $\mathtt{Step} \prec_{\mathcal{F}} \mathtt{Eval}$ where each symbol has now a product status. It has a quasi-interpretation:

$$(\!|\mathbf{c}|\!)(S, \Pi_1, \cdots, \Pi_m) = S + \sum_i \Pi_i + 1$$
$$(\!|\mathtt{Step}|\!)(X) = X + 1$$
$$(\!|\mathtt{Eval}|\!)(T, X) = T + X$$

□

Unlike the previous proof, this simulation can not be extended in order to capture parallel computation.

8.4 Plugging time

It remains to compute the clock. This is done by the following rules,

Proposition 62 *Polynomial (respectively exponential and double exponential) functions are computed by additive (resp. affine, multiplicative) RPO^{QI}_{Lin}-programs.*

PROOF.

$$\mathtt{add}(\diamond, y) \to y \quad \mathtt{add}(\mathbf{s}(x), y) \to \mathbf{s}(\mathtt{add}(x, y)) \tag{6}$$
$$\mathtt{mult}(\diamond, y) \to \diamond \quad \mathtt{mult}(\mathbf{s}(x), y) \to \mathtt{add}(y, \mathtt{mult}(x, y)) \tag{7}$$
$$\mathtt{exp}(\diamond) \to \mathbf{s}(\diamond) \quad \mathtt{exp}(\mathbf{s}'(x)) \to \mathtt{add}(\mathtt{exp}(x), \mathtt{exp}(x)) \tag{8}$$
$$\mathtt{dexp}(\diamond) \to \mathbf{s}(\mathbf{s}(\diamond)) \quad \mathtt{dexp}(\mathbf{s}'(x)) \to \mathtt{mult}(\mathtt{dexp}(x), \mathtt{dexp}(x)) \tag{9}$$

These rules form a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program by putting $\mathtt{add} \prec_{\mathcal{F}} \mathtt{mult} \prec_{\mathcal{F}} \{\mathtt{exp}, \mathtt{dexp}\}$ and with the quasi-interpretations for function symbols

$$\begin{aligned}
(\!|\mathtt{add}|\!)(X,Y) &\; X+Y \\
(\!|\mathtt{mult}|\!)(X,Y) &= (X+1)\times(Y+1) \\
(\!|\mathtt{exp}|\!)(X) &= X+1 \\
(\!|\mathtt{dexp}|\!)(X) &= X+2
\end{aligned}$$

and for constructors

	additive	affine	multiplicative
$(\!\|\diamond\|\!)$	0	0	0
$(\!\|\mathtt{s}\|\!)(X)$	$X+1$	$X+1$	$X+1$
$(\!\|\mathtt{s}'\|\!)(X)$		$2X+1$	$(X+3)^2$
Rules interpreted	$6-7$	$6-7-8$	$6-7-8-9$

□

Remark 63 *Notice that, given the quasi-interpretations for* $\boldsymbol{s}$ *and* $\boldsymbol{s}'$*, it is not possible to write a program admitting a polynomial quasi-interpretation (whatever the kind) which transforms a tally integer written with* $\boldsymbol{s}$ *into one written with* $\boldsymbol{s}'$*. The reason lies on the fact that both class of function* Ptime *and* Etime *are distinct.*

8.5 Sum-up of the simulations

Theorem 64

(1) A polynomial time function is computed by an additive RPO^{QI}_{Lin}*-program (resp.* RPO^{QI}_{Pro}*-program or* $RPO^{QI}_{Pro+Lin}$*-program).*
(2) A exponential time function is computed by an affine RPO^{QI}_{Lin}*-program (resp.* RPO^{QI}_{Pro}*-program or* $RPO^{QI}_{Pro+Lin}$*-program).*
(3) A doubly exponential time function is computed by a multiplicative RPO^{QI}_{Lin}*-program (resp.* RPO^{QI}_{Pro}*-program or* $RPO^{QI}_{Pro+Lin}$*-program).*

Theorem 65

(1) A polynomial space function is computed by an additive RPO^{QI}*-program.*
(2) A exponential space function is computed by an affine RPO^{QI}*-program.*

(3) A doubly exponential space function is computed by a multiplicative RPO^{QI}-program.

References

[1] R. Amadio. Max-plus quasi-interpretations. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings*, volume 2701 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2003.

[2] R.M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, 2004. to appear.

[3] N. Andersen and N.D. Jones. Generalizing Cook's transformation to imperative stack programs. In J. Karhumäki, H. Maurer, and G. Rozenberg, editors, *Results and trends in theoretical computer science*, volume 812 of *Lecture Notes in Computer Science*, pages 1–18, 1994.

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

[5] G. Bonfante. *Constructions d'ordres, analyse de la complexité*. Thèse, Institut National Polytechnique de Lorraine, 2000.

[6] 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.

[7] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11, 2000.

[8] V.-H. Caseiro. *Equations for defining Poly-Time*. PhD thesis, University of Oslo, 1997.

[9] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[10] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of computer Programming*, pages 131–159, 1987.

[11] 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.

[12] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *2nd GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, 1975.

[13] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving termination of rewriting with CiME. In *wst'03*, pages 71–73, 2003. wst'03, `http://cime.lri.fr`.

[14] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, January 1971.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

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

[17] J. Giesl. Generating polynomial orderings for termination proofs. In *RTA*, number 914 in Lecture Notes in Computer Science, pages 427–431, 1995.

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

[19] 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.

[20] 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.

[21] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[22] N. D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228:151–174, 1999.

[23] S. Kamin and J-J Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerity of Illinois, Urbana, 1980. Unpublished note. Accessible on http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.

[24] M. S. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theoretical Computer Science*, 40(2-3):323–328, 1985.

[25] D.S. Lankford. On proving term rewriting systems are noetherien. Technical Report Accessible on http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html, 1979.

[26] 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.

[27] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1,2):167,184, September 1993.

[28] 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.

[29] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 1992.

[30] J.-Y. Marion. *Complexité implicite des calculs, de la théorie la pratique*. Habilitation à diriger les recherches, Université Nancy 2, 2000.

[31] J.-Y. Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183:2–18, 2003.

[32] J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, Nov 2000.

[33] Y. V. Matiyasevich. *Hilbert's 10th Problem*. Foundations of Computing Series. The MIT Press, 1993. MAT y 93:1 1.Ex.

[34] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report R-78-943, Department of Computer Science, University of Illinois, 1978. Accessible on http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.

[35] W. J. Savitch. Relationship between nondeterministic and deterministic tape classes. *JCSS*, 4:177–192, 1970.

[36] G. Senizergues. Some undecidable termination problems for semi-thue systems. *Theoretical Computer Science*, 142:257–276, 1995.

[37] A. Tarski. *A Decision Method for Elementary Algebra and Geometry, 2nd ed.* University of California Press, 1951.

[38] A. Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.