Theoretical Computer Science 412 (2011) 2776–2796

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Quasi-interpretations a way to control resources

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

[a] Nancy-Universtité, Loria, INPL-ENSMN, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France

[b] LIPN, Université Paris 13, 99, Avenue J-B Clément, 93430 Villetaneuse, France

ARTICLE INFO

Article history:
Received 1 June 2004
Received in revised form 24 January 2011
Accepted 4 February 2011
Communicated by D. Sannella

Keywords:
Implicit complexity
Quasi-interpretation
Termination orderings

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 allow the analysis of system complexity. A quasi-interpretation is a numerical assignment, which provides an upper bound on computed functions and which is compatible with the program operational semantics. The quasi-interpretation method offers several advantages: (i) It provides hints in order to optimize an execution, (ii) it gives resource certificates, and (iii) finding quasi-interpretations is decidable for a broad class which is relevant for feasible computations.

By combining the quasi-interpretation method with termination tools (here term orderings), we 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 potentially to any formalism that can be reduced to transition systems. A quasi-interpretation gives a kind of measure 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 constructor term is a real which bounds its size. Second, a quasi-interpretation weakly decreases when a term is reduced.

From a practical point of view, the quasi-interpretation method is a tool to perform complexity analysis in a static way. Quasi-interpretations allow one 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 which consists in the (partial) proof of the fact that a program admits a quasi-interpretation.

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 **Max-Poly** quasi-interpretation which is built by combining the max operator of fixed arity and polynomials of bounded degrees. This leads to an automatic synthesis procedure of a meaningful class of quasi-interpretations.

From a theoretical point of view, we combine quasi-interpretations with termination tools. We focus on simplification orderings and we consider in particular the Recursive Path Orderings introduced by Dershowitz [16]. It turns out that we characterize the class Ptime of functions computable in polynomial time [31] and the class Pspace of functions computable in polynomial space [8].

This work is related to the ideas of Cobham [12], Bellantoni and Cook [5], Leivant [25] and Leivant–Marion [26] for delineating complexity classes. Note that most of the machine-independent characterizations of complexity classes have

* Corresponding author. Tel.: +33 149404443; fax: +33 148260712.
E-mail addresses: bonfante@loria.fr (G. Bonfante), Jean-Yves.Marion@loria.fr (J.-Y. Marion), Jean-Yves.Moyen@lipn.univ-paris13.fr (J.-Y. Moyen).

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2011.02.007

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. In this long-term research program, the completeness problem 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 an intentional computability theory. Similar questions have been raised by Caseiro [10], Hofmann [20] and Jones [22]. It is also worth mentioning the studies on intentionality of Colson, see for example [13], and of Moschovakis, as well as Gurevich.

Lastly, Marion and Péchoux suggested a new method, called sup-interpretation [32], which is closely related to quasi-interpretations. Sup-interpretations allow one to capture more algorithms, and to characterize small parallel complexity classes [9]. On the other hand, sup-interpretations do not have the nice properties of quasi-interpretations.

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 that 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 with respect to a broad class of polynomially bounded assignments. Section 5 defines the 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 using call by value semantics with a cache in order to have an exponential speed-up. The last Section 8 is devoted to simulations of both space and time bounded computations.

2. First order functional programming

Term rewriting systems underpin first order functional programming, that is why we refer to the survey of Dershowitz and Jouannaud [17]. Throughout the following discussion, we consider three finite 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)	$\mathcal{T}(\mathcal{C}) \ni v$	$::= \mathbf{c} \mid \mathbf{c}(v_1, \ldots, v_n)$
(terms)	$\mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t$	$::= \mathbf{c} \mid x \mid \mathbf{c}(t_1, \ldots, t_n) \mid \mathtt{f}(t_1, \ldots, t_n)$
(patterns)	$\mathcal{P} \ni p$	$::= \mathbf{c} \mid x \mid \mathbf{c}(p_1, \ldots, p_n)$
(rules)	$\mathcal{D} \ni d$	$::= \mathtt{f}(p_1, \ldots, p_n) \to t$

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$. We shall use a typewriter 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 finite 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 $\mathtt{f}$.

The set of rules induces a rewriting relation $\to$. The relation $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$. 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 $\mathtt{f}(p_1, \ldots, p_n) \to t$ is left-linear, that is a variable appears only once in $\mathtt{f}(p_1, \ldots, p_n)$, and (ii) there are no two left-hand sides which are overlapping. Lastly, a ground term is a term with no variables.

2.2. Semantics

Orthogonal programs define a class of deterministic first order functional programs. The domain of the computed functions is the constructor term algebra $\mathcal{T}(\mathcal{C})$.

A substitution σ is a mapping from variables to terms. We say that it is a constructor substitution when the range of σ is $\mathcal{T}(\mathcal{C})$. We denote $\mathfrak{S}$ as the set of these constructor substitutions.

We consider a call by value semantics which is displayed in Fig. 1. The meaning of $t \downarrow w$ is that t evaluates to a constructor term w. The program $\mathtt{f}$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ defined as follows. For every $v_1, \ldots, v_n \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{f}]\!](v_1, \ldots, v_n) = w$ iff $\mathtt{f}(v_1, \ldots, v_n) \downarrow w$. Otherwise, it is undefined and $[\![\mathtt{f}]\!](v_1, \ldots, v_n) = \bot$.

Notice that if $t \downarrow w$ then $t \xrightarrow{*} w$, because programs are confluent.

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow w_i}{\mathbf{c}(t_1, \ldots, t_n) \downarrow \mathbf{c}(w_1, \ldots, w_n)} \; (Constructor)$$

$$\frac{t_i \downarrow w_i \quad \mathtt{f}(p_1, \ldots, p_n) \rightarrow r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = w_i \quad r\sigma \downarrow w}{\mathtt{f}(t_1, \ldots, t_n) \downarrow w} \; (Function)$$

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

3. Quasi-interpretations

3.1. Quasi-interpretation definition

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 Marion [29,30], Bonfante [6], and Marion–Moyen [31]. There are related to interpretation to prove termination, and in particular to [7].

The fundamental property of a quasi-interpretation is that it is a numerical approximation from above of the size of each intermediate value (that is constructor terms), which appears in a reduction process. However, a quasi-interpretation does not give an upper bound on the term size that appears in a reduction process. A typical example is the `lcs` example 6 whose reduction involved terms of exponential size, but the `lcs` program admits an additive quasi-interpretation, as we shall see later.

Let $\mathbb{R}^+$ be the set of non-negative real numbers. An assignment $(\!| - |\!)$ is a mapping from constructors and function symbols, that is $\mathcal{C} \bigcup \mathcal{F}$, such that for each symbol b of arity n it yields

(1) An non-negative real number $(\!| c |\!)$ of $\mathbb{R}^+$, for every symbol c of arity 0.
(2) a n-ary function $(\!| b |\!) : (\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$ for every symbol b of arity $n > 0$.

Take a denumerable sequence $X_1, \ldots, X_n, \ldots$. We extend an assignment $(\!| - |\!)$ to terms canonically. Given a term t with n variables $x_1, \ldots, x_n$, the assignment $(\!| t |\!)$ denotes a function from $(\mathbb{R}^+)^n$ to $\mathbb{R}^+$ and is defined as follows:

$$(\!| x_i |\!) = X_i \qquad\qquad x_i \in \mathcal{X}$$
$$(\!| b(t_1, \ldots, t_n) |\!) = (\!| b |\!)((\!| t_1 |\!), \ldots, (\!| t_n |\!))$$

An assignment satisfies the *subterm property* if for any $i = 1, n$ and any $X_1, \ldots, X_n$ in $\mathbb{R}^+$, we have

$$(\!| b |\!)(X_1, \ldots, X_n) \geq X_i \tag{1}$$

A direct consequence of the subterm property is that for any ground term s and any subterm t of s, $(\!| s |\!) \geq (\!| t |\!)$.

An assignment is *weakly monotone* if for any symbol b, $(\!| b |\!)$ is an increasing (not necessarily strictly) function with respect to each variable. That is, for every symbol b and for all $i = 1, n$ if $X_i \geq Y_i$, we have $(\!| b |\!)(X_1, \ldots, X_n) \geq (\!| b |\!)(Y_1, \ldots, Y_n)$.

A substitution σ is defined over a term t, if the domain of σ contains all variables of t. Given two terms t and u, we say that $(\!| t |\!) \geq (\!| u |\!)$ if for every constructor substitution σ defined over t and u, we have $(\!| t\sigma |\!) \geq (\!| u\sigma |\!)$.

Definition 3 (*Quasi-interpretation*)**.** A quasi-interpretation $(\!| - |\!)$ of a program `f` is a weakly monotonic assignment satisfying the subterm property such that for each rule $l \rightarrow r$

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

Throughout, when we write "quasi-interpretation", we always mean "quasi-interpretation of a program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$".

It is worth noticing that the inequalities that define a quasi-interpretation are not strict, which differs from the notion of interpretation used to prove termination.

Proposition 4. *Assume that $(\!| - |\!)$ is a quasi-interpretation of a program f. For any ground terms u and v such that $u \xrightarrow{*} v$, we have $(\!| u |\!) \geq (\!| v |\!)$.*

Proof. A context is a particular term that we write $\mathtt{C}[\diamond]$, where $\diamond$ is a new variable. The substitution of $\diamond$ in $\mathtt{C}[\diamond]$ by a term t is denoted $\mathtt{C}[t]$.

The proof goes by induction on the derivation length n. For this, suppose that $u = u_0 \rightarrow \cdots \rightarrow u_n = v$. If $n = 0$ then the result is immediate. Otherwise $n > 0$ and in this case, there is a rule $\mathtt{f}(p_1, \ldots, p_n) \rightarrow t$ and a constructor substitution σ such that $u_0 = \mathtt{C}[\mathtt{f}(p_1, \ldots, p_n)\sigma]$ and $u_1 = \mathtt{C}[t\sigma]$. Since $(\!| - |\!)$ is a quasi-interpretation, we have $(\!| t\sigma |\!) \leq (\!| \mathtt{f}(p_1, \ldots, p_n)\sigma |\!)$. The weak monotonicity property implies that $(\!| \mathtt{C}[t\sigma] |\!) \leq (\!| \mathtt{C}[\mathtt{f}(p_1, \ldots, p_n)\sigma] |\!)$. We conclude by induction hypothesis. $\square$

Example 5. Given a list l of tally natural numbers, $\mathtt{sort}(l)$ sorts the elements of l by insertion. The constructor set is $\mathcal{C} = \{\mathbf{tt}, \mathbf{ff}, \mathbf{0}, \mathbf{suc}, \mathbf{nil}, \mathbf{cons}\}$.

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

Constructors admit the following quasi-interpretation.

$$
\begin{aligned}
(\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) = (\!|\mathbf{0}|\!) = (\!|\mathbf{nil}|\!) &= 0 \\
(\!|\mathbf{suc}|\!)(X) &= X + 1 \\
(\!|\mathbf{cons}|\!)(X, Y) &= X + Y + 1
\end{aligned}
$$

And function symbols

$$
\begin{aligned}
(\!|\mathtt{if\ \ then\ \ else}|\!)(X, Y, Z) &= \max(X, Y, Z) \\
(\!|<|\!)(X, Y) &= \max(X, Y) \\
(\!|\mathtt{insert}|\!)(X, Y) &= X + Y + 1 \\
(\!|\mathtt{sort}|\!)(X) &= X
\end{aligned}
$$

This example illustrates two important facts. Quasi-interpretations can be max-functions like in the case of $<$. And, 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 $\mathbf{suc}^4(\mathbf{0})$ because the length longest common subsequence is 4 (take **baba**).

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

It admits the following quasi-interpretation:

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

3.2. Taxonomy of quasi-interpretations

Our aim is to study feasible computations. That is why we confine ourselves to programs admitting quasi-interpretations which are bounded by polynomials. We insist that assignments are bounded by polynomials, but are not necessarily polynomials.

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.

Next, we classify polynomial assignment according to the rate of growth of constructor assignments.

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

- An assignment of **c** is *additive* (or of kind 0) if

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

where $\alpha \geq 1$.
- An assignment of **c** is *affine* (or of kind 1) if

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

where the β_i's are constants of $\mathbb{R}^+$ and $\alpha \geq 1$.
- An assignment **c** is *multiplicative* (or of kind 2) if

$$(\!|\mathbf{c}|\!)(X_1, \ldots, X_n) = Q(X_1, \ldots, X_n) + \alpha$$

where Q is a polynomial and $\alpha \geq 1$.

We classify *polynomial* assignments by the kind of assignments given to constructors, and not to function symbols. If each constructor assignment is additive (resp. affine, multiplicative) then the assignment is an additive (resp. affine, multiplicative) assignment.

A program `f` admits an additive quasi-interpretation $(\!|\_|\!)$ if it is an additive assignment. We shall also just say that `f` is additive, without explicitly mentioning the additive quasi-interpretation tied to it.

Similarly, a program `f` admits an affine (resp. a multiplicative) quasi-interpretation $(\!|\_|\!)$ if it is an affine (resp. a multiplicative) assignment. We shall also just say that `f` is affine (resp. multiplicative).

In both previous examples, programs admit a polynomial quasi-interpretation because each quasi-interpretation is bounded by a polynomial. In Example 5, the quasi-interpretation of the function symbol $<$ is not a polynomial. The insertion sort program admits an additive quasi-interpretation because each constructor (that is the symbol in {**tt**, **ff**, **0**, **suc**, **nil**, **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.

Example 9. We give three programs that illustrate the three kinds of program classes delineated by quasi-interpretations.

$$\mathtt{add}(\mathbf{0}, y) \to y \tag{2}$$
$$\mathtt{add}(\mathbf{suc}(x), y) \to \mathbf{suc}(\mathtt{add}(x, y)) \tag{3}$$
$$\mathtt{mult}(\mathbf{0}, y) \to \mathbf{0} \tag{4}$$
$$\mathtt{mult}(\mathbf{suc}(x), y) \to \mathtt{add}(y, \mathtt{mult}(x, y)) \tag{5}$$

These rules define the addition and the multiplication. They admit the following additive quasi-interpretation.

$$(\!|\mathbf{0}|\!) = 0 \tag{6}$$
$$(\!|\mathbf{suc}|\!)(X) = X + 1 \tag{7}$$
$$(\!|\mathtt{add}|\!)(X, Y) = X + Y \tag{8}$$
$$(\!|\mathtt{mult}|\!)(X, Y) = X \times Y \tag{9}$$

So, addition and multiplication are additive programs (additivity only refers to the interpretation of constructors). Now, in order to define the exponential, we introduce another successor **s** which has an affine assignment.

$$\mathtt{exp}(\mathbf{0}) \to \mathbf{suc}(\mathbf{0}) \tag{10}$$
$$\mathtt{exp}(\mathbf{s}(x)) \to \mathtt{add}(\mathtt{exp}(x), \mathtt{exp}(x)) \tag{11}$$

The quasi-interpretations of the new symbols are:

$$(\!|\mathbf{s}|\!)(X) = 2X + 1 \tag{12}$$
$$(\!|\mathtt{exp}|\!)(X) = X + 1 \tag{13}$$

The above program which defines the exponential admits an affine quasi-interpretation. We see that the domain of `exp` and its co-domain are not the same. Indeed, the domain is generated by {**0**, **s**} whose quasi-interpretation is affine and the co-domain is generated by {**0**, **suc**} whose quasi-interpretation is additive. We shall see later on that it is necessary to have two successors with different kinds of quasi-interpretations. Similar observations have be done in [7] and on tiering system in which an argument of tier 1 produces an output of tier 0. We think that it is worth investigating this analogy in order to interpret tiering concepts by quasi-interpretations.

We define the doubly-exponential function, i.e. $n \mapsto 2^{2^n}$, as follows (here we need yet another successor, $\mathbf{s}'$).

$$\mathtt{dexp}(\mathbf{0}) \to \mathbf{suc}(\mathbf{suc}(\mathbf{0})) \tag{14}$$

$$\mathtt{dexp}(\mathbf{s}'(x)) \to \mathtt{mult}(\mathtt{dexp}(x), \mathtt{dexp}(x)) \tag{15}$$

and

$$(\!|\mathbf{s}'|\!)(X) = (X+2)^2 \tag{16}$$

$$(\!|\mathtt{dexp}|\!)(X) = X + 2 \tag{17}$$

Again we see that the domain and co-domain are not the same. The domain admits a multiplicative quasi-interpretation and the co-domain has an additive one.

Other classes of assignments could be introduced, such as elementary or primitive recursive assignments, but we will not discuss them in this paper. This type of extension is related to Lescanne's paper [28] about interpretation for termination proofs.

3.3. Elementary properties of assignments

We study now some quantitative properties of assignments when they are of the kind mentioned above. The size $|t|$ of a term t is defined by

$$|t| = \begin{cases} 0 & \text{if } t \text{ is 0-ary symbol} \\ 1 + \sum_{i=1,n} |t_i| & \text{if } t = b(t_1, \ldots, t_n) \end{cases}$$

Proposition 10. *Assume that $(\!|\_|\!)$ is an additive, an affine or a multiplicative assignment. For any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $|t| \leq (\!|t|\!)$.*

Proof. The proof goes by induction on the size of t. $\square$

Proposition 11. *Assume that $(\!|\_|\!)$ is an additive, an affine or a multiplicative quasi-interpretation of a program f. For any term u and any constructor term $t \in \mathcal{T}(\mathcal{C})$, if $u \xrightarrow{*} t$, we have $|t| \leq (\!|u|\!)$.*

Proof. Proposition 4 implies that $(\!|u|\!) \geq (\!|t|\!)$. Then from Proposition 10, we have $|t| \leq (\!|t|\!)$. So, $|t| \leq (\!|u|\!)$. $\square$

Proposition 12.

- *If $(\!|\_|\!)$ is an additive assignment, for any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $(\!|t|\!) \leq k \times |t|$.*
- *If $(\!|\_|\!)$ is an affine assignment, for any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $(\!|t|\!) \leq 2^{k \times |t|}$.*
- *If f is a multiplicative program, for any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $(\!|t|\!) \leq 2^{2^{k \times |t|}}$*

where in each case k is a constant which depends on the assignment $(\!|\_|\!)$ given to constructors.

Proof. The proof goes by induction on the size of t. $\square$

It is worth noticing that the above Proposition illustrates a general phenomenon that we shall see throughout 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.

3.4. Call-trees

We now present call-trees which are a tool that we shall use frequently. A call-tree gives a static view of an execution that captures all function calls. Hence, we can study dependencies between function calls without worrying about the extra details provided by the underlying rewriting relation.

Take a program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$. A *state* of a program $\mathtt{f}$ is a tuple $\langle \mathtt{h}, v_1, \ldots, v_p \rangle$, where $\mathtt{h}$ is a function symbol of $\mathcal{F}$ of arity p and $v_1, \ldots, v_p$ are constructor terms of $\mathcal{T}(\mathcal{C})$. Throughout, we may omit to mention the program $\mathtt{f}$ when the context is clear.

Assume that $\eta_1 = \langle \mathtt{h}, v_1, \ldots, v_p \rangle$ and $\eta_2 = \langle \mathtt{g}, u_1, \ldots, u_m \rangle$ are two states. A *transition* is a triplet $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ such that:

(i) e is a rule $\mathtt{h}(q_1, \ldots, q_p) \to t$ of $\mathcal{E}$,
(ii) there is a constructor substitution σ such that $q_i\sigma = v_i$ for all $1 \leq i \leq p$,
(iii) there is a subterm $\mathtt{g}(s_1, \ldots, s_m)$ of t such that for any $1 \leq i \leq m$, $s_i\sigma \xrightarrow{*} u_i$ and $u_i \in \mathcal{T}(\mathcal{C})$.

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

Proposition 13. *Let $(\!|-|\!)$ be a quasi-interpretation of a program f. Assume that $\langle h, v_1, \ldots, v_p\rangle$ and $\langle g, u_1, \ldots, u_m\rangle$ are two states such that*

$$\langle h, v_1, \ldots, v_p\rangle \overset{*}{\rightsquigarrow} \langle g, u_1, \ldots, u_m\rangle$$

Then we have $(\!|g(u_1, \ldots, u_m)|\!) \le (\!|h(v_1, \ldots, v_p)|\!)$ and also $(\!|u_i|\!) \le (\!|h(v_1, \ldots, v_p)|\!)$ for all $1 \le i \le m$.

Proof. The hypothesis $\langle \mathtt{h}, v_1, \ldots, v_p\rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, u_1, \ldots, u_m\rangle$ means that there is a term t such that $\mathtt{h}(v_1, \ldots, v_p) \xrightarrow{*} t$ and that $\mathtt{g}(u_1, \ldots, u_m)$ is a subterm of t. Proposition 4 states $(\!|\mathtt{h}(v_1, \ldots, v_p)|\!) \ge (\!|t|\!)$. Since a quasi-interpretation satisfies the subterm property, we have $(\!|u_i|\!) \le (\!|\mathtt{g}(u_1, \ldots, u_m)|\!) \le (\!|\mathtt{h}(v_1, \ldots, v_p)|\!)$. □

Next, we define the $\langle \mathtt{g}, u_1, \ldots, u_m\rangle$-call tree as a tree where (i) $\langle \mathtt{g}, u_1, \ldots, u_m\rangle$ is the root. (ii) the set of nodes is $\{\eta \mid \langle \mathtt{g}, u_1, \ldots, u_m\rangle \overset{*}{\rightsquigarrow} \eta\}$ and (iii) there is an edge between the state η_1 and the state η_2 if $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$.

Some states may actually appear several times in the tree. This happen typically in two cases: when there is a loop in the computation or when there are two different sequences of call leading to the same one (such as with the Fibonacci function). We do not merge identical states in the call-tree, hence nodes are occurrences of a state rather that a state alone.

Keeping several occurrences of the same state is useful because here we need to mimic the call by value semantics of Fig. 1. This semantics actually performs identical calls several times. In Section 7 identical nodes in a call tree will be merged and the tree will thus turn into a directed acyclic graph.

The size of a state $\langle \mathtt{g}, u_1, \ldots, u_m\rangle$ is $\sum_{i=1}^{m} |u_i|$.

Lemma 14. *Let $(\!|-|\!)$ be an additive (or affine, or multiplicative) quasi-interpretation of a program f. The size of each node of the $\langle f, t_1, \ldots, t_n\rangle$-call graph is bounded by $d \times (\!|f(t_1, \ldots, t_n)|\!)$, where d is the maximal arity of a function symbol in f.*

Proof. Suppose that $\langle \mathtt{g}, u_1, \ldots, u_m\rangle$ is a state of the $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$-call graph. It follows from Proposition 13 that $(\!|u_i|\!) \le (\!|\mathtt{f}(t_1, \ldots, t_n)|\!)$. As each u_i is a constructor term, Proposition 10 entails that $|u_i| \le (\!|u_i|\!)$. Therefore

$$|\langle \mathtt{g}, u_1, \ldots, u_m\rangle| = \sum_{i=1,n} |u_i| \le d \times (\!|\mathtt{f}(t_1, \ldots, t_n)|\!)$$

where d is the maximal arity of a function symbol. □

3.5. Upper bound on the complexity

It turns out that we can now state a quite important practical point. Indeed, consider an additive program $\mathtt{f}$. Then, the quasi-interpretation of $(\!|\mathtt{f}(t_1, \ldots, t_n)|\!)$ is bounded by a polynomial in the input size, that is in $\sum_{i=1,n}(|t_i|)$. Next, by combining Lemma 14, we deduce that the size of each state of the $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$-call tree is bounded by a polynomial in the input size, because the size of each state is bounded by the quasi-interpretation of the root.

Theorem 15. *Assume that f is a program. For any constructor terms $t_1, \ldots, t_n$,*

- *If f is an additive program, the size of each state of the $\langle f, t_1, \ldots, t_n\rangle$-call tree is bounded by $P(m)$, where P is some polynomial.*
- *If f is an affine program, the size of each state of the $\langle f, t_1, \ldots, t_n\rangle$-call tree is bounded by $2^{k\times m}$, where k is some constant.*
- *If f is a multiplicative program, the size of each state of the $\langle f, t_1, \ldots, t_n\rangle$-call tree is bounded by $2^{2^{k\times m}}$, where k is some constant*

where $m = \max_{i=1,n} |t_i|$.

Proof. It is a consequence of Lemma 14 and Proposition 12. □

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. In [1], Amadio wrote a first proof of the result above.

Theorem 16. *There is an evaluation procedure which, given an additive program f and given n constructor terms $t_1, \ldots, t_n$, computes the value w if $f(t_1, \ldots, t_n) \downarrow w$ and otherwise returns $\perp$, that is if the evaluation does not terminate. This evaluation procedure runs in exponential time, i.e. in $2^{P(\max_{i=1}^{n} |t_i|)}$, where P is a polynomial.*

Proof. We build a call by value evaluator with a deterministic Turing machine with an extra tape that behaves as a stack in order to evaluate $\mathtt{f}(t_1, \ldots, t_n)$. The stack is used for the recursive calls and the normal tapes contain the current context. Actually, a call by value procedure computes the value of each state of $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$-call graph and for this we perform breadth-first exploration. A context corresponds to a state and the number of states that we have to memorize is bounded by the width of the call-graph. Also, the width is bounded by the maximal arity d of a function symbol. So, the space use on the tape is bounded by $k' \times d \times (\!|\mathtt{f}(t_1, \ldots, t_n)|\!)$ for some constant k'. (Notice that, here, we do not take into consideration the size of the stack.) Cook's theorem [14] implies that the call by value evaluator can be then simulated in time $2^{K\times (\!|\mathtt{f}(t_1,\ldots,t_n)|\!)}$ for some constant K (which depends on k' and d). Since the program admits a polynomial quasi-interpretation, the time is bounded by $2^{P(\max_{i=1}^{n} |t_i|)}$, where $P(X) = K \times (\!|\mathtt{f}|\!)(kX, \ldots, kX)$ by Proposition 12. □

Corollary 17. *There is an evaluation procedure which given an affine (resp. multiplicative) program f and n constructor terms $t_1, \ldots, t_n$, computes the value w if $f(t_1, \ldots, t_n) \downarrow w$ and otherwise returns $\perp$ in double exponential time, i.e. in $2^{2^{K'\times \max_{i=1}^{n} |t_i|}}$ (resp. in triple exponential time, i.e. in $2^{2^{2^{K''\times \max_{i=1}^{n} |t_i|}}}$), where K' and K'' are two constants.*

3.6. 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 18. *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 conclusion follows immediately. □

4. Synthesis of quasi-interpretations

We consider now the problem of finding program quasi-interpretations, which is an important practical question. For this, we restrict assignments to the class **Max-Poly**. The class of **Max-Poly** functions contains constant functions ranging over non-negative rationals and is closed by projections, maximum, addition, multiplication and composition.

We establish as a direct consequence of Tarski's Theorem [37] that finding a program quasi-interpretation, which belongs to the class **Max-Poly** over non-negative real numbers, is decidable, when degrees are fixed. Indeed, Tarski demonstrated that the first-order theory for reals containing the addition $+$, the multiplication $\times$, the equality $=$, the order $>$ with variables over reals and rational constants is decidable. On the other hand, the same question over natural numbers is undecidable because it is a consequence of Matiyasevich's Theorem [33].

We consider two related problems. The first one is the *verification problem*:

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

The second one is the *synthesis problem*:

input: A program $\mathtt{f}$.
problem: Is there an assignment $(\!|-|\!)$ which is a quasi-interpretation for $\mathtt{f}$?

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

Proposition 19 (*Normalization*)**.** *A **Max-Poly** function Q can always be expressed as:*

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

where each P_i is a polynomial. We say that the max-*degree of Q is k and the degree of Q is the maximum degree of the polynomials $P_1, \dots, P_k$.*

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

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

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

where $(\!|l|\!) = \max(P_1, \dots, P_n)$, $(\!|r|\!) = \max(Q_1, \dots, Q_m)$ and $X_1, \dots, X_p$ 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 20. *The verification problem for **Max-Poly** assignments is decidable in exponential time in the size of the program.*

Proof. In order to solve the verification problem, we have to decide whether or not the following first order formula is true.

$$S_{\mathcal{E}} = \bigwedge_{l\to r\in\mathcal{E}} S_{l\to r} \quad \text{for a given assignment}$$

This is performed by Tarski's decision procedure. Basu et al. [4] established that such procedure is at most exponential in the number of quantifiers. In our case, it corresponds to the maximum arity of symbols. □

Theorem 21. *The synthesis problem for **Max-Poly** assignment of bounded degree and bounded* max-*degree is decidable in doubly 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 of the form

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

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

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

where $S_{\mathcal{E}}$ is defined in the previous proof. 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 22. 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. 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 the **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 is related to non-size increasing according to Hofmann [19]. Amadio established that the synthesis of the 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-interpretations. 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] and Kamin and Lévy [23]. Finally, Krishnamoorthy and Narendran in [24] have proved that deciding whether 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 23. The product extension[1] of $\prec$ over sequences, denoted $\prec^p$, is defined as follows.
We have $(m_1, \ldots, m_k) \prec^p (n_1, \ldots, n_k)$ if and only if (i) $\forall i \leq k$, $m_i \preceq n_i$ and (ii) $\exists j \leq k$ such that $m_j \prec n_j$.

Definition 24. The lexicographic extension of $\prec$, denoted $\prec^l$, is defined as follows.
We have $(m_1, \ldots, m_k) \prec^l (n_1, \ldots, n_k)$ 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 ordering of sequences in a restriction of the more usual multi-set ordering of sequences. We do not need here the full power of the multi-set ordering mainly because we only compare sequences of the same length while the multi-set ordering works on sequences of any length.

Notice that the product ordering of sequences is also a restriction of the lexicographic ordering, that is two sequences ordered by the product extension are also ordered lexicographically.

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 $\mathcal{F}$.

[1] Unlike [31], 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.

$$\frac{u = t_i \text{ or } u \prec_{rpo} t_i}{u \prec_{rpo} \mathtt{f}(\ldots, t_i, \ldots)} \ \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{\forall i\, u_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathbf{c}(u_1, \ldots, u_m) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)} \ \mathtt{f} \in \mathcal{F}, \mathbf{c} \in \mathcal{C}$$

$$\frac{\forall i\, u_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{\mathtt{g}(u_1, \ldots, u_m) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

$$\frac{(u_1, \ldots, u_n) \prec_{rpo}^{st(\mathtt{f})} (t_1, \ldots, t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i\, u_i \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)}{\mathtt{g}(u_1, \ldots, u_n) \prec_{rpo} \mathtt{f}(t_1, \ldots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

Fig. 2. Definition of $\prec_{rpo}$.

Definition 25. 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 statuses are compatible with precedences.

Definition 26. Given a precedence $\preceq_{\mathcal{F}}$ and a status st, the recursive path ordering $\prec_{rpo}$ is defined in Fig. 2.

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

Theorem 27 (*Dershowitz [16]*). *Each program which is ordered by $\prec_{rpo}$ terminates on all inputs.*

Example 28.

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

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

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

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

(3) The program `sort` of Example 5 terminates if each function symbol has a product status and by setting the precedence `if then else` $\prec_{\mathcal{F}}$ `insert` $\prec_{\mathcal{F}}$ `sort`.

(4) The `lcs` program of Example 6 is ordered by taking `max` $\prec_{\mathcal{F}}$ `lcs`, and both symbols have a product status.

5.3. Extensional characterization

The orderings considered are special cases of 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[2]

Theorem 29. *The set of functions computed by $\mathrm{RPO}_{\mathrm{Pro}}$-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[3]

Theorem 30. *The set of functions computed by $\mathrm{RPO}_{\mathrm{Lex}}$-programs is exactly the set of multiple-recursive functions.*

2 Since the product ordering is a restriction of the multiset ordering, any $\mathrm{RPO}_{\mathrm{Pro}}$-program is also terminating by the more usual MPO termination ordering. Conversely, as stated above, there is a (somewhat tedious) procedure to turn MPO programs into $\mathrm{RPO}_{\mathrm{Pro}}$-programs.

3 Here, the $\mathrm{RPO}_{\mathrm{Lex}}$-programs are exactly the programs terminating by the usual LPO termination ordering.

5.4. Consequences of termination proofs

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

Proposition 31.

(1) *For each constructor term t and u, $u \prec_{rpo} t$ iff $u \triangleleft t$.*
(2) *For each constructor term $u_1, \ldots, u_n$ and $t_1, \ldots, t_n$, $(u_1, \ldots, u_n) \prec^x_{rpo} (t_1, \ldots, t_n)$ implies $(u_1, \ldots, u_n) \triangleleft^x (t_1, \ldots, t_n)$, where x is a status p or l and $\triangleleft^x$ is the corresponding extension based on the subterm relation.*
(3) *For each constructor term $u_1, \ldots, u_n$ and $t_1, \ldots, t_n$, $(u_1, \ldots, u_n) \prec^x_{rpo} (t_1, \ldots, t_n)$ implies $(|u_1|, \ldots, |u_n|) <^x (|t_1|, \ldots, |t_n|)$, 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. □

Remark 32. The ordering $\prec_{rpo}$ is not stable for constructor contexts. Indeed, we have $\mathbf{1}(\epsilon) \prec_{rpo} \mathbf{0}(\mathbf{1}(\epsilon))$, but $\mathbf{1}(\mathbf{1}(\epsilon)) \prec_{rpo} \mathbf{1}(\mathbf{0}(\mathbf{1}(\epsilon)))$ does not hold. So, $\prec_{rpo}$ is not a reduction ordering but there is no rewriting inside constructor terms.

Lemma 33. *Let 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 $\langle f, t_1, \ldots, t_n\rangle$-call tree is strictly bounded by A. Then the following facts hold:*

(1) *If $\langle f, t_1, \ldots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, u_1, \ldots, u_m\rangle$ then*
   (a) *$g \prec_{\mathcal{F}} f$ or*
   (b) *$g \approx_{\mathcal{F}} f$ and $(u_1, \ldots, u_m) \prec^{st(f)}_{rpo} (t_1, \ldots, t_n)$.*
(2) *If $\langle f, t_1, \ldots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, u_1, \ldots, u_m\rangle$ and $g \approx_{\mathcal{F}} f$ then the number of states between the states $\langle f, t_1, \ldots, t_n\rangle$ and $\langle g, u_1, \ldots, u_m\rangle$ 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 $\prec_{rpo}$.
(2) Suppose that $\langle \mathtt{h}, v_1, \ldots, v_p\rangle$ is a state between $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$ and $\langle \mathtt{g}, u_1, \ldots, u_m\rangle$. Due to the first point of this lemma, we have $\mathtt{h} \approx_{\mathcal{F}} \mathtt{f}$ and $(v_1, \ldots, v_p) \prec^{st(\mathtt{f})}_{rpo} (t_1, \ldots, t_n)$. So, by Proposition 31(3), we have $(|v_1|, \ldots, |v_p|) <^{st(\mathtt{f})} (|t_1|, \cdots, |t_n|)$. 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 symbols 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 34. A RPO^{QI}-program is a program that (i) admits a quasi-interpretation and (ii) terminates by $\prec_{rpo}$.

Theorem 35. *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 38. The completeness of this characterization is established by Theorem 55.

Example 36. 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 ¬, ∨, ∃. The QBF problem is solved by the following program.

$$\begin{array}{rr}
\mathtt{not}(\mathbf{tt}) \to \mathbf{ff} & \mathtt{not}(\mathbf{ff}) \to \mathbf{tt} \\
\mathtt{or}(\mathbf{tt}, x) \to \mathbf{tt} & \mathtt{or}(\mathbf{ff}, x) \to x \\
\mathbf{0} = \mathbf{0} \to \mathbf{tt} & \mathbf{suc}(x) = \mathbf{0} \to \mathbf{ff} \\
\mathbf{0} = \mathbf{suc}(y) \to \mathbf{ff} & \mathbf{suc}(x) = \mathbf{suc}(y) \to x = y \\
\mathtt{in}(x, \mathbf{nil}) \to \mathbf{ff} & \mathtt{in}(x, \mathbf{cons}(a, l)) \to \mathtt{or}(x = a, \mathtt{in}(x, l))
\end{array}$$

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

Booleans are encoded by $\{\mathbf{tt}, \mathbf{ff}\}$, variables are encoded by unary integers which are generated by $\{\mathbf{0}, \mathbf{suc}\}$. 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

$$\{\texttt{not}, \texttt{or}, \_\texttt{=}\_\} \prec_{\mathcal{F}} \texttt{in} \prec_{\mathcal{F}} \texttt{verify} \prec_{\mathcal{F}} \texttt{qbf}$$

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

They admit the following additive quasi-interpretations:

$$\begin{aligned}
(\!|\mathbf{c}|\!) &= 0 && \text{where } \mathbf{c} \text{ is a constructor of arity } 0 \\
(\!|\mathbf{c}|\!)(X_1, \ldots, X_n) &= 1 + \sum_{i=1}^{n} X_i && \text{where } \mathbf{c} \text{ is a constructor of arity} > 0 \\
(\!|\texttt{verify}|\!)(\Phi, T) &= \Phi + T \\
(\!|\texttt{qbf}|\!)(\Phi) &= \Phi + 1 \\
(\!|\texttt{g}|\!)(X_1, \ldots, X_n) &= \max_{i=1}^{n} X_i && \text{for other function symbols}
\end{aligned}$$

6.2. *RPO^{QI}-programs are* Pspace *computable*

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

Lemma 37. *Let f be a RPO^{QI}-program. For each constructor term $t_1, \ldots, t_n$, the space used by a call by value interpreter to compute $f(t_1, \ldots, t_n)$ is bounded by a polynomial in $(\!|f(t_1, \ldots, t_n)|\!)$.*

Proof. Take an innermost call by value interpreter, like the one of Fig. 1. It builds recursively in a depth first manner the $\langle \texttt{f}, t_1, \ldots, t_n\rangle$-call tree, evaluates nodes and backtracks. Put $A = (\!|\texttt{f}(t_1, \ldots, t_n)|\!)$. 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 33(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 $O(A^{d+1})$. □

Theorem 38. *Let f be an additive RPO^{QI}-program. For each constructor term $t_1, \ldots, t_n$, the space used by a call by value interpreter to compute $f(t_1, \ldots, t_n)$ is bounded by a polynomial in $\max_{i=1}^{n} |t_i|$.*

Proof. By Proposition 12, we have $(\!|t_i|\!) \leq O(|t_i|)$. Because quasi-interpretations are polynomially bounded, we have $(\!|\texttt{f}(t_1, \ldots, t_n)|\!) \leq P(\max_{i=1}^{n} |t_i|)$, for some polynomial P. So the space is bounded by $O(P(\max_{i=1}^{n} |t_i|)^{d+1})$ following Lemma 37. □

6.3. *Beyond polynomial space*

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

Theorem 39.

- *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 38. The kind of quasi-interpretation gives the different upper-bounds on the space-usage as established in Proposition 12. The converse is established by Theorems 58 and 61.

7. Characterizing time bounded computation

7.1. Polynomial time computation

Definition 40. A function symbol f is linear in a program terminating by $\prec_{rpo}$ if for each rule $\mathtt{f}(p_1, \ldots, p_n) \rightarrow r$, then there is at most one occurrence in r of a function symbol g with the same precedence as f, that is $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$.

Definition 41.

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

Tail recursive programs are $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-programs, as is illustrated by the `reverse` program in Example 28. On the other hand, the program that solves QBF in Example 36 is not a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program, because of the definition of `verify` (in the case of **Exists**(n, ϕ)), 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 [27] by tiering discipline.

Theorem 42. *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 47 below. The completeness of this characterization is established by Theorem 54.

Example 43. 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 Fig. 4. Hence, we simulate dynamic programming techniques, which consist in storing each result of a function call in a table and avoiding the recomputation of the same function call if it is already in the table. This technique is inspired from Andersen and Jones' re-reading [3] of Cook simulation technique over 2-way push-down automata [14] and is called memoization.

The expression $\langle C, t\rangle \Downarrow \langle C', w\rangle$ means that the computation of t is w 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 $(\mathtt{g}, w_1, \ldots, w_m, w)$ where $\langle \mathtt{g}, w_1, \ldots, w_m\rangle$ is a state, and $[\![\mathtt{g}]\!](w_1, \ldots, w_m) = w$. When a term $\mathtt{g}(w_1, \ldots, w_m)$ is considered, we search for a configuration $(\mathtt{g}, w_1, \ldots, w_m, w)$ in the current cache C. If such configuration exists, we use it to short-cut the computation and so we return w. Otherwise, we apply a program equation, say $l \rightarrow r$, by matching $\mathtt{g}(w_1, \ldots, w_m)$ with l. Then, we update C by adding the configuration $(\mathtt{g}, w_1, \ldots, w_m, w)$ to the current cache C.

Fig. 3 shows what happens to the $\langle$`lcs`, **ababa**, **baaba**$\rangle$-call tree when memoization 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 44. *Suppose that $\langle C_0, f(t_1, \ldots, t_n)\rangle \Downarrow \langle C, w\rangle$. The size of the final cache C is bounded by a polynomial in $(\!| f(t_1, \ldots, t_n) |\!)$.*

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

$$\#C = \sum_{\mathtt{g}\in\mathcal{F}} \#C_{\mathtt{g}} \tag{18}$$

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

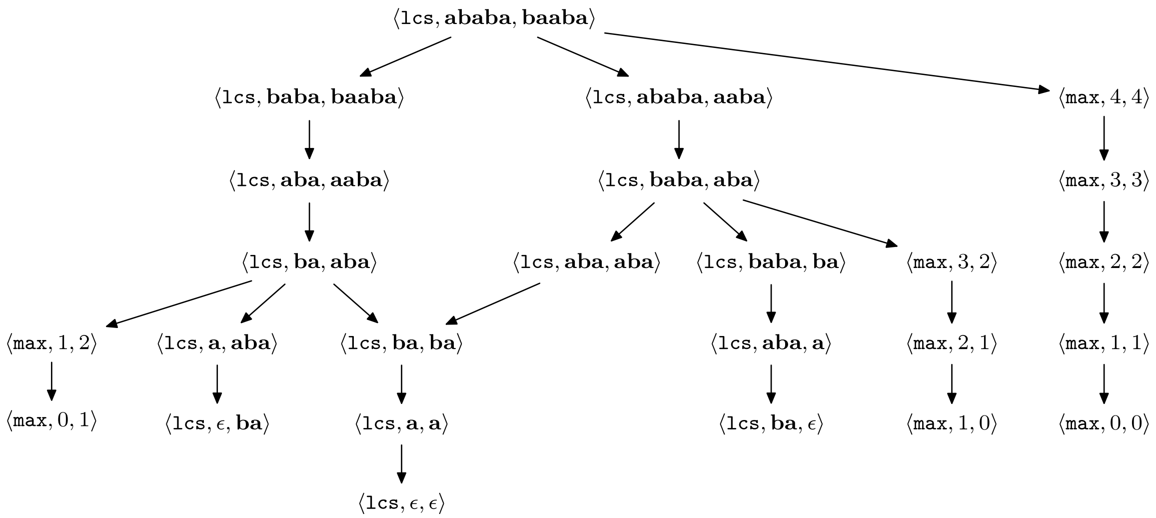

Fig. 3. The $\langle \texttt{lcs}, \textbf{ababa}, \textbf{baaba}\rangle$-call tree with memoization.

(Constructor)

$$\frac{\mathbf{c} \in \mathcal{C} \quad \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, w_i\rangle}{\langle C_0, \mathbf{c}(t_1, \ldots, t_n)\rangle \Downarrow \langle C_n, \mathbf{c}(w_1, \ldots, w_n)\rangle}$$

(Read)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, w_i\rangle \quad (\texttt{f}, w_1, \ldots, w_n, w) \in C_n}{\langle C_0, \texttt{f}(t_1, \ldots, t_n)\rangle \Downarrow \langle C_n, w\rangle}$$

(Update)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, w_i\rangle \quad \texttt{f}(p_1, \ldots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = w_i \quad \langle C_n, r\sigma\rangle \Downarrow \langle C, w\rangle}{\langle C_0, \texttt{f}(t_1, \ldots, t_n)\rangle \Downarrow \langle C \cup (\texttt{f}, w_1, \ldots, w_n, w), w\rangle}$$

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

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

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

$$\#C^{\vee}_{\texttt{g}} \leq \sum_{\texttt{g} \prec_{\mathcal{F}} \texttt{f}} \#C_{\texttt{f}} \tag{19}$$

- We consider $C^{\wedge}_{\texttt{g}}$.
  (1) The status of g is product. Proposition 31 states that sub-calls of the same rank starting from $\texttt{f}(v_1, \ldots, v_n)$ 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 14 that the number of sub-calls is bounded

$$\prod_{i \leq n}(|v_i| + 1) \leq (d \times (\!| \texttt{f}(t_1, \ldots, t_n) |\!))^d \tag{20}$$

  where d is the maximal arity of a function symbol.

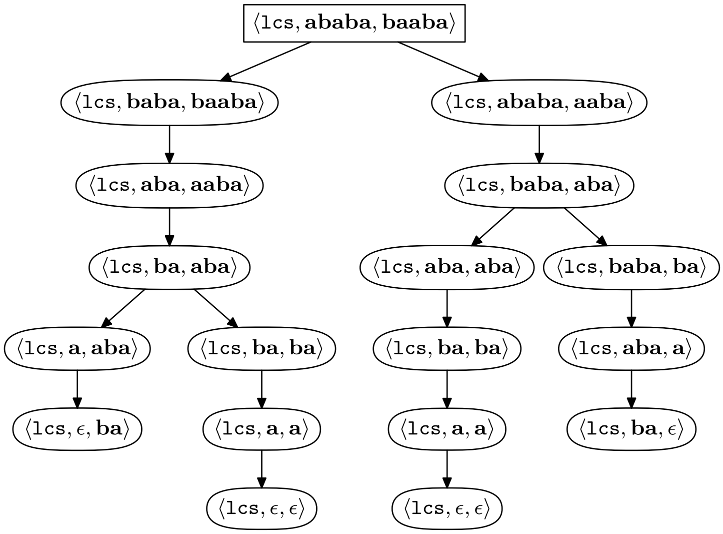

Fig. 5. The `lcs`-covering graph of the call-tree.

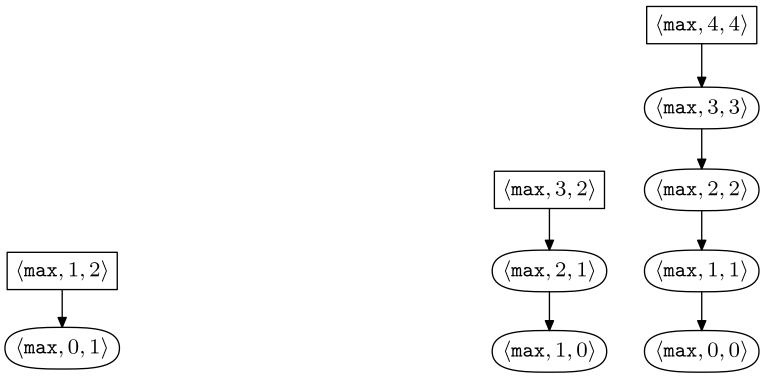

Fig. 6. The `max`-covering graph of the call-tree.

(2) The status of g is lexicographic. But by definition of $\text{RPO}^{\text{QI}}_{\text{Lin}}$-programs, there is at most one recursive call starting from g for each rule application. Lemma 33(3) entails that the maximal length of a branch is $(\!|\mathtt{f}(t_1, \dots, t_n)|\!)^d$, which is also a bound on the number of successive calls initiated by `f`.

From both previous points, we obtain that

$$\#C^{\wedge}_{\mathtt{g}} \leq (\#C^{\vee}_{\mathtt{g}} + 1) \times d^d \times (\!|\mathtt{f}(t_1, \dots, t_n)|\!)^d \tag{21}$$

Finally, we have

$$\#C_{\mathtt{g}} \leq \#C^{\vee}_{\mathtt{g}} + \#C^{\wedge}_{\mathtt{g}} \tag{22}$$

By combining (18), (19), (21), and (22), we see that the cardinality of C is polynomially bounded in $(\!|\mathtt{f}(t_1, \dots, t_n)|\!)$. □

Example 45. Figs. 5 and 6 show the covering graphs of `lcs` and `max` in the ⟨`lcs`, **ababa**, **baaba**⟩-call tree. Nodes in $C^{\vee}_{\mathtt{g}}$ are shown as squares while nodes in $C^{\wedge}_{\mathtt{g}}$ are shown as circles ($\mathtt{g} \in \{\mathtt{lcs}, \mathtt{max}\}$).

Lemma 46. *Let f be a $RPO^{QI}_{Pro+Lin}$-program. For each constructor term $t_1, \dots, t_n$, the runtime of the call by value interpreter with cache to compute $f(t_1, \dots, t_n)$ is bounded by a polynomial in $(\!|f(t_1, \dots, t_n)|\!)$.*

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

Proof. By Proposition 12, we have $(\!|t_i|\!) \leq O(|t_i|)$. For some polynomial P we have $(\!|\mathtt{f}(t_1, \ldots, t_n)|\!) \leq P(\max_{i=1}^n |t_i|)$, because quasi-interpretations are polynomially bounded. Lemma 46 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 [29].

We see that if a function symbol is linear, see Definition 40, then no result needs to be recorded. More generally, consider the $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$-call tree. When evaluating $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$, the call by value semantics with cache stores all values. Say that a separation set N is a set of states such that each chain starting from the root state $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$ 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 $\langle \mathtt{f}, t_1, \ldots, t_n\rangle$ 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 set. We can require an implementation to keep a minimal separation set. To perform this dynamically, we have to compare configurations in the cache. Take two configurations $(\mathtt{f}, t_1, \ldots, t_n, t)$ and $(\mathtt{g}, u_1, \ldots, u_m, u)$. If $\mathtt{f}(t_1, \ldots, t_n) \prec_{rpo} \mathtt{g}(u_1, \ldots, u_m)$ then we no longer need the configuration $(\mathtt{f}, t_1, \ldots, t_n, t)$ and we can erase it from the cache.

7.3. Beyond polynomial time

Theorem 48.

- *The set of functions computed by* affine $RPO^{QI}_{Pro+Lin}$*-programs (resp.* RPO^{QI}_{Pro}*-programs and* RPO^{QI}_{Lin}*-programs) 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}*-programs and* RPO^{QI}_{Lin}*-programs) 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 47. The kind of quasi-interpretation gives the different upper-bounds on the time-usage as established in Proposition 12. The converse is again a consequence of Theorems 57 and 60.

8. Simulation of Parallel Register Machines

8.1. Parallel register machines

Following [27], we introduce *Parallel Register Machines (PRM)*, which are able to model the essential features of both traditional sequential computing, like Turing Machines, and alternating computations, like Alternating Turing Machines.

A PRM M works over the word algebra $\mathbb{W}$ generated by the constructors $\{\mathbf{0}, \mathbf{1}, \boldsymbol{\epsilon}\}$ and 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
$[\mathbf{Succ}(\pi = \mathbf{i}(\pi), s')]$, $[\mathbf{Pred}(\pi = \mathbf{p}(\pi), s')]$, $[\mathbf{Branch}(\pi, s', s'')]$,
$[\mathbf{Fork}_{\min}(s', s'')]$, $[\mathbf{Fork}_{\max}(s', s'')]$, $[\mathbf{End}]$.

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 values. We denote $\{\pi \leftarrow \pi'\}F$ to mean that the value of the register π is the content of π', the other registers stay unchanged.

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 $\blacktriangleleft$.

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

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 $(w_1, \dots, w_k) \in \mathbb{W}^k$, we have

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

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

8.2. Space and time bounded computation

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

Proposition 49. *A function ϕ is computable in polynomial (respectively exponential, doubly exponential) time iff ϕ is RM-computable in polynomial time (resp. exponential, doubly exponential).*

There are pleasingly well-known tight connections between space used by a Turing machine and time used by PRM. The essence of the translation comes from the work of Savitch [35] and Chandra et al. [11].

Theorem 50. *A function ϕ is computable in polynomial (resp. exponential, doubly exponential) space iff ϕ is PRM-computable in polynomial time (resp. exponential, doubly exponential).*

8.3. Time bounded simulation with lexicographic termination

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

Lemma 51 (*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*

$$\begin{array}{llll} f: & \mathbb{N} \times \mathbb{W} & \to \mathbb{W} & \\ & (n, w) & \mapsto \phi(w) & \text{if } n > T(|w|) \\ & (n, w) & \mapsto \bot & \text{otherwise} \end{array}$$

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. Suppose that f is computed by a PRM M. The simulation of the PRM M is done by following the rules of the semantic partial function $\mathtt{Eval}$. 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 first that $\min_{\blacktriangleleft}$ and $\max_{\blacktriangleleft}$ are additive RPO^{QI}-program.

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

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

We associate the following quasi-interpretations:

$$\begin{array}{ccc}
(\!|\boldsymbol{\epsilon}|\!) = 0 & (\!|\diamond|\!) = 0 & \forall q \in S, (\!|q|\!) = 0 \\
(\!|\mathbf{0}|\!)(X) = X + 1 & (\!|\mathbf{1}|\!)(X) = X + 1 & (\!|\mathbf{s}|\!)(X) = X + 1 \\
(\!|\mathtt{min}|\!)(W, W') = \max(W, W') & (\!|\mathtt{max}|\!)(W, W') = \max(W, W') &
\end{array}$$

Next, we write a program to compute the semantic partial function $\mathtt{Eval}$.

(a) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \dots, \mathbf{i}(\pi_j), \dots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')$,
(b) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \mathbf{i}(\pi_j), \dots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \dots, \pi_j, \cdots, \pi_m)$
if $com(s) = \mathbf{Pred}(\pi_j = \mathbf{p}(\pi_j), s')$,
(c) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \mathbf{0}(\pi_j), \cdots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1, \dots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$,
(d) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \dots, \mathbf{1}(\pi_j), \cdots, \pi_m) \to \mathtt{Eval}(t, s'', \pi_1, \dots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$,

(e) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \texttt{min}(\texttt{Eval}(t, s', \pi_1, \ldots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\min}(s', s'')$,
(f) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \texttt{max}(\texttt{Eval}(t, s', \pi_1, \ldots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \ldots, \pi_m))$
if $com(s) = \mathbf{Fork}_{\max}(s', s'')$,
(g) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \to \pi_m$, if $com(s) = \mathbf{End}$.

Finally, put $\texttt{f}(t, w) \to \texttt{Eval}(t, \textsc{begin}, w, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon})$. It is routine to check that $f = [\![\texttt{f}]\!]$.
These programs admit the following quasi-interpretations:

$$(\!|\texttt{Eval}|\!)(T, S, \Pi_1, \ldots, \Pi_m) = T + S + \sum_{i=1}^{m} \Pi_i$$
$$(\!|\texttt{f}|\!)(T, X) = T + X$$

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$^{\mathrm{QI}}_{\mathrm{Lin}}$-program if f is computed by a RM. □

8.4. Time bounded simulation with product termination

In [31], the simulation of RM is performed by a RPO$^{\mathrm{QI}}_{\mathrm{Pro}}$-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 52 (*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*

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

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

Proof. Compared with the previous proof, the simulation is performed in a bottom-up way. For this, we use an extra constructor **c** to encode tuples. And we define Step which gives the next configuration.

(a) $\texttt{Step}(\mathbf{c}(s, \pi_1, \ldots, \pi_m)) \to \mathbf{c}(s', \pi_1 \cdots, \mathbf{i}(\pi_j), \ldots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')$
(b) $\texttt{Step}(\mathbf{c}(s, \pi_1, \ldots, \mathbf{i}(\pi_j), \ldots, \pi_m)) \to \mathbf{c}(s', \pi_1, \ldots, \pi_j, \cdots, \pi_m)$
if $com(s) = \mathbf{Pred}(\pi_j = \mathbf{p}(\pi_j), s')$
(c) $\texttt{Step}(\mathbf{c}(s, \pi_1, \ldots, \mathbf{0}(\pi_j), \ldots, \pi_m) \to \mathbf{c}(s', \pi_1, \ldots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$
(d) $\texttt{Step}(\mathbf{c}(s, \pi_1, \ldots, \mathbf{1}(\pi_j), \ldots, \pi_m) \to \mathbf{c}(s'', \pi_1, \ldots, \pi_m)$
if $com(s) = \mathbf{Branch}(\pi_j, s', s'')$
(e) $\texttt{Step}(\mathbf{c}(s, \pi_1, \ldots, \pi_m) \to \mathbf{c}(s, \pi_1, \ldots, \pi_m)$
if $com(s) = \mathbf{End}$

The simulation is made by

$$\begin{aligned} \texttt{Eval}(\boldsymbol{\epsilon}, x) &\to x \\ \texttt{Eval}(\mathbf{s}(t), x) &\to \texttt{Step}(\texttt{Eval}(t, x)) \\ \texttt{f}(t, w) &\to \texttt{Eval}(t, \mathbf{c}(\textsc{begin}, w, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon})) \end{aligned}$$

The rules are ordered by putting $\texttt{Step} \prec_{\mathcal{F}} \texttt{Eval}$, where each symbol has now a product status. A quasi-interpretation of the rules is

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

The other constructor assignments are identical to the ones in the previous simulation described in the proof of Lemma 51

$$(\!|\boldsymbol{\epsilon}|\!) = 0 \qquad (\!|\diamond|\!) = 0 \qquad \forall q \in S, (\!|q|\!) = 0$$
$$(\!|\mathbf{0}|\!)(X) = X + 1 \quad (\!|\mathbf{1}|\!)(X) = X + 1 \quad (\!|\mathbf{s}|\!)(X) = X + 1 \quad \square$$

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

Now, it remains to compute the clock, that is the length of the iteration of the main loop of the simulation, in order to complete the simulation.

8.5. Simulation of polynomial computations

Proposition 53. *Any polynomial is computed by an additive RPO^{QI}_{Lin}-programs (resp. RPO^{QI}_{Pro}-programs).*

Proof. We define any polynomial by composition from the additive programs for the addition add and the multiplication mult.

$$\begin{aligned}
\mathtt{add}(\diamond, y) &\to y\\
\mathtt{add}(\mathbf{s}(x), y) &\to \mathbf{s}(\mathtt{add}(x, y))\\
\mathtt{mult}(\diamond, y) &\to \diamond\\
\mathtt{mult}(\mathbf{s}(x), y) &\to \mathtt{add}(y, \mathtt{mult}(x, y))
\end{aligned}$$

The programs add and mult admit the following quasi-interpretations

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

where constructors have the quasi-interpretation

$$\begin{aligned}
(\!|\diamond|\!) &= 0\\
(\!|\mathbf{s}|\!)(X) &= X + 1
\end{aligned}$$

The programs add and mult terminate by $\prec_{rpo}$ by putting $\mathtt{add} \prec_{\mathcal{F}} \mathtt{mult}$ with a product status. So, any polynomial is a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program. On the other hand, add and mult are linear and thus any polynomial is also a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program. □

Theorem 54.

- *A polynomial time function is computed by an additive RPO^{QI}_{Lin}-program.*
- *A polynomial time function is computed by an additive RPO^{QI}_{Pro}-program.*
- *A polynomial time function is computed by an additive $RPO^{QI}_{Pro+Lin}$-program.*

Proof. Let ϕ be a function which is computed by a RM in time bounded by a polynomial P.

For the first case, the time bound P is computed by an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program following Proposition 53. We compose with Lemma 51, we conclude that ϕ is computed by an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program.

For the second case, the time bound P is also an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program following Proposition 53. We compose with Lemma 52, we conclude that ϕ is computed by an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program.

The last case is an immediate consequence of the previous constructions. □

Theorem 55. *A polynomial space function is computed by an additive RPO^{QI}-program.*

Proof. Let ϕ be a function which is computed by a PRM in time bounded by a polynomial P. Since P is also computed by a RPO^{QI}-program following Proposition 53 and by composing with Lemma 51, we conclude that ϕ is computed by an additive RPO^{QI}-program. □

8.6. Simulation of exponential computations

Proposition 56. *Let $\gamma > 0$ be a constant. The function $\lambda n.2^{\gamma n}$ is computed by an affine RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program).*

Proof. The function $\lambda n.2^{\gamma n}$ is computed by

$$\begin{aligned}
\mathtt{mk}_\gamma(\diamond) &\to \diamond\\
\mathtt{mk}_\gamma(\mathbf{s}'(x)) &\to \tilde{\mathbf{s}}'(\dots(\tilde{\mathbf{s}}'(\mathtt{mk}_\gamma(x)))\dots) && \gamma \text{ times}\\
\mathtt{d}(x) &\to \mathtt{add}(x, x) && \mathtt{add} \text{ is defined in Proposition 53}\\
\mathtt{exp}(\diamond) &\to \mathbf{s}(\diamond)\\
\mathtt{exp}(\tilde{\mathbf{s}}'(x)) &\to \mathtt{d}(\mathtt{exp}(x))\\
\mathtt{exp}_\gamma(x) &\to \mathtt{exp}(\mathtt{mk}_\gamma(x))
\end{aligned}$$

We have $[\![\mathtt{exp}_\gamma]\!](n) = m$, where $n = (\mathbf{s}')^n(\diamond)$ and $m = (\mathbf{s})^{2^{\gamma n}}(\diamond)$.

Constructors have the following quasi-interpretation

$$
\begin{aligned}
(\!|\diamond|\!) &= 0\\
(\!|\mathbf{s}|\!)(X) &= X+1\\
(\!|\mathbf{s}'|\!)(X) &= 2^{\gamma}X+2^{\gamma}-1\\
(\!|\tilde{\mathbf{s}}|\!)(X) &= 2X+1
\end{aligned}
$$

And this program admits the following quasi-interpretations

$$
\begin{aligned}
(\!|\mathtt{mk}_k|\!)(X) &= X\\
(\!|\mathtt{d}|\!)(X) &= 2X\\
(\!|\mathtt{exp}|\!)(X) &= X+1\\
(\!|\mathtt{exp}_{\gamma}|\!)(X) &= X+1
\end{aligned}
$$

This program terminates by $\prec_{rpo}$ with a product status. So, $\lambda n.2^{\gamma n}$ is a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program and also a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program. □

Theorem 57. *A exponential time function is computed by an affine RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program or $RPO^{QI}_{Pro+Lin}$-program).*

Theorem 58. *An exponential space function is computed by an affine RPO^{QI}-program.*

8.7. Simulation of doubly exponential computations

Proposition 59. *Let $\gamma > 0$ be a constant. The function $\lambda n.2^{2^{\gamma n}}$ is computed by an multiplicative RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program).*

Proof. The function $\lambda n.2^{2^{\gamma n}}$ is computed by

$$
\begin{aligned}
\mathtt{dmk}_{\gamma}(\diamond) &\to \diamond\\
\mathtt{dmk}_{\gamma}(\mathbf{s}''(x)) &\to \tilde{\mathbf{s}}''(\dots(\tilde{\mathbf{s}}''(\mathtt{dmk}_{\gamma}(x)))\dots) && \gamma \text{ times}\\
\mathtt{square}(x) &\to \mathtt{mult}(x,x) && \mathtt{mult} \text{ is defined in Proposition 53}\\
\mathtt{dexp}(\diamond) &\to \mathbf{s}(\mathbf{s}(\diamond))\\
\mathtt{dexp}(\tilde{\mathbf{s}}''(x)) &\to \mathtt{square}(\mathtt{dexp}(x))\\
\mathtt{dexp}_{\gamma}(x) &\to \mathtt{dexp}(\mathtt{dmk}_{\gamma}(x))
\end{aligned}
$$

We have $[\![\mathtt{dexp}_{\gamma}]\!](n) = m$, where $n = (\mathbf{s}'')^n(\diamond)$ and $m = (\mathbf{s})^{2^{2^{\gamma n}}}(\diamond)$.

Constructors have the following quasi-interpretation

$$
\begin{aligned}
(\!|\diamond|\!) &= 0\\
(\!|\mathbf{s}|\!)(X) &= X+1\\
(\!|\mathbf{s}''|\!)(X) &= \theta_{\gamma}(X)\\
(\!|\tilde{\mathbf{s}}''|\!)(X) &= (X+2)^2
\end{aligned}
$$

where

$$
\begin{aligned}
\theta_0(X) &= X\\
\theta_{k+1}(X) &= (\theta_k(X)+2)^2 \quad 0 \le k < \gamma
\end{aligned}
$$

And the program admits the following quasi-interpretations

$$
\begin{aligned}
(\!|\mathtt{dmk}_{\gamma}|\!)(X) &= X\\
(\!|\mathtt{square}|\!)(X) &= X^2\\
(\!|\mathtt{dexp}|\!)(X) &= X+2\\
(\!|\mathtt{dexp}_{\gamma}|\!)(X) &= X+2
\end{aligned}
$$

This program terminates by $\prec_{rpo}$ with a product status. So, $\lambda n.2^{2^{\gamma n}}$ is a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program and also a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program. □

Theorem 60. *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 61. *A doubly exponential space function is computed by a multiplicative RPO^{QI}-program.*

References

[1] R. Amadio, Max-plus quasi-interpretations, in: Martin Hofmann (Ed.), Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10–12, 2003, Proceedings, in: Lecture Notes in Computer Science, vol. 2701, Springer, 2003, pp. 31–45.

[2] R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, L. Jakubiec, A functional scenario for bytecode verification of resource bounds, in: Jerzy Marcinkowski, Andrzej Tarlecki (Eds.), Computer Science Logic, 18th International Workshop, CSL 13th Annual Conference of the EACSL, Karpacz, Poland, in: Lecture Notes in Computer Science, vol. 3210, Springer, 2004, pp. 265–279.

[3] N. Andersen, N.D. Jones, Generalizing Cook's transformation to imperative stack programs, in: J. Karhumäki, H. Maurer, G. Rozenberg (Eds.), Results and Trends in Theoretical Computer Science, in: Lecture Notes in Computer Science, vol. 812, 1994, pp. 1–18.

[4] S. Basu, R. Pollack, M.-F. Roy, On the combinatorial and algebraic complexity of quantifier elimination, Journal of the ACM 43 (6) (1996) 1002–1045.

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

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

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

[8] G. Bonfante, J.-Y. Marion, J.-Y. Moyen, On lexicographic termination ordering with space bound certifications, in: Dines Bjørner, Manfred Broy, Alexandre V. Zamulin (Eds.), Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, Ershov Memorial Conference, in: Lecture Notes in Computer Science, vol. 2244, Springer, 2001.

[9] Guillaume Bonfante, Jean-Yves Marion, Romain Péchoux, A characterization of alternating log time by first order functional programs, in: LPAR, in: Lecture Notes in Computer Science, vol. 4246, Springer, 2006, pp. 90–104.

[10] V.-H. Caseiro, Equations for defining Poly-Time, Ph.D. Thesis, University of Oslo, 1997.

[11] A. Chandra, D. Kozen, L. Stockmeyer, Alternation, Journal of the ACM 28 (1981) 114–133.

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

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

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

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

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

[17] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: Handbook of Theoretical Computer Science, vol. B, Elsevier Science Publishers B. V., North-Holland, 1990, pp. 243–320.

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

[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, 1999, pp. 464–473.

[20] M. Hofmann, A type system for bounded space and functional in-place update, in: European Symposium on Programming, ESOP'00, in: Lecture Notes in Computer Science, vol. 1782, 2000, pp. 165–179.

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

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

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

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

[25] D. Leivant, Predicative recurrence and computational complexity I: word recurrence and poly-time, in: Peter Clote, Jeffery Remmel (Eds.), Feasible Mathematics II, Birkhäuser, 1994, pp. 320–343.

[26] D. Leivant, J.-Y. Marion, Lambda calculus characterizations of poly-time, Fundamenta Informaticae 19 (1–2) (1993) 167–184.

[27] D. Leivant, J.-Y. Marion, Ramified recurrence and computational complexity II: substitution and poly-space, in: L. Pacholski, J. Tiuryn (Eds.), Computer Science Logic, 8th Workshop, CSL'94, in: Lecture Notes in Computer Science, vol. 933, Springer, Kazimierz, Poland, 1995, pp. 486–500.

[28] P. Lescanne, Termination of rewrite systems by elementary interpretations, in: H. Kirchner, G. Levi (Eds.), 3rd International Conference on Algebraic and Logic Programming, in: Lecture Notes in Computer Science, vol. 632, Springer, 1992, pp. 21–36.

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

[30] J.-Y. Marion, Analysing the implicit complexity of programs, Information and Computation 183 (2003) 2–18. Presented at ICC workshop affiliated with Floc 1999, Trento.

[31] J.-Y. Marion, J.-Y. Moyen, Efficient first order functional program interpreter with time bound certifications, in: Michel Parigot, Andrei Voronkov (Eds.), Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, in: Lecture Notes in Computer Science, vol. 1955, Springer, 2000, pp. 25–42.

[32] J.-Y. Marion, R. Péchoux, Resource analysis by sup-interpretation, in: FLOPS, in: Lecture Notes in Computer Science, vol. 3945, Springer, 2006, pp. 163–176.

[33] Y.V. Matiyasevich, Hilbert's 10th Problem, in: Foundations of Computing Series, The MIT Press, 1993.

[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.

[35] W.J. Savitch, Relationship between nondeterministic and deterministic tape classes, Journal of Computer System Science 4 (1970) 177–192.

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

[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 (1995) 335–362.