A characterization of polynomial complexity classes using dependency pairs

Jean-Yves Marion, Romain Péchoux

A characterization of polynomial complexity classes using dependency pairs

Jean-Yves Marion and Romain Péchoux

Nancy-Université, Loria, Carte team, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France, and École Nationale Supérieure des Mines de Nancy, INPL, France.
Jean-Yves.Marion@loria.fr Romain.Pechoux@loria.fr

Abstract. The dependency pair method has already shown its power in proving termination of term rewriting systems. We adapt this framework using polynomial assignments in order to characterize with two distinct criteria the set of the functions computable in polynomial time and the set of the functions computable in polynomial space. To our knowledge, this is a first attempt to capture complexity classes using of the dependency pair method. The characterizations presented are inspired by previous works on implicit computational complexity, and, particularly, by the notions of quasi-interpretation and sup-interpretation. Both criteria are decidable so that we can synthesize resource upper-bounds.

1 Introduction

The dependency pair method has been introduced in [1] in order to prove the termination of programs automatically. The dependency pair method is a complete termination criterion. In other words, the termination of any terminating program or, more precisely, of any term rewriting system, can be demonstrated with the help of such a criterion. However, undecidability of termination forces to specialize this method through applications. Nowadays, almost all termination provers, among others AProVE [10], CiME [8] and TTT [11], take advantage of such applications.

One of the key challenges is to improve these termination tools in order to analyze the computational complexity of programs automatically. Indeed, there is a strong relation between termination and computational complexity. In order to predict the complexity of the function computed by a program, we first have to prove that it terminates and, then, to put drastic restrictions on either its syntax or its semantics. The motivations behind such an analysis are to provide resource certificates by static analysis. For example, a resource certificate can give the guarantee to the programmer that no buffer overflow occurs during the execution of a program.

Throughout the paper, we focus our attention on the dependency pair method using polynomial assignments. Polynomial assignments are strongly related to polynomial interpretations of [15, 17] and, more recently, of [2, 7]. However, we no longer consider polynomials over natural numbers but polynomials over real

numbers. Since the set of real numbers is no longer well-founded, we can add constraints on the shape of considered polynomials for preserving termination. The subterm property of [9], i.e. $\forall i \in \{1, n\}\ \mathtt{f}(x_1, \cdots, x_n) > x_i$, is an example of such a constraint. Nevertheless, this condition is a serious drawback when analyzing the complexity of algorithms since it approximates functions too largely. Hopefully, it is demonstrated in [16] that this condition can be replaced by a suitable quasi-ordering. Polynomial assignments are also inspired by quasi-interpretations [5] and sup-interpretations [18]. These two notions provide upper bounds on the size of the values computed by a function symbol. The sup-interpretation is only a generalization of the quasi-interpretation without the subterm property and, as a consequence, allows to capture more algorithms. It is demonstrated in [5, 19] that these notions combined with the product extension and the lexicographic extension of Recursive Path Orderings (RPO) characterize the set of functions computable in polynomial time and, respectively, the set of functions computable in polynomial space. Moreover, sup-interpretations allow to characterize the set of functions computable in alternating logarithmic time [4].

In this paper, we make a step forward in the study of relations between computational complexity and termination by giving two new characterizations of the sets of functions computable in polynomial time and in polynomial space with the help of polynomial assignments over reals without the subterm property. To our knowledge, this is a first attempt to capture complexity classes using of the dependency pair method. As demonstrated by Tarski, first order theory over reals is decidable and so are our criteria whenever we consider polynomials of bounded degree. Another consequence is that we obtain heuristics for synthesizing sup-interpretations.

The paper is organized as follows. Section 2 describes the syntax and the semantics of the language. Section 3 introduces the dependency pairs and dependency pair graph. Section 4 defines the polynomial assignments which are used in the characterizations. Sections 5 and 6 provide two distinct criteria which correspond to the characterization of the set of function computable in polynomial space and, respectively, the set of functions computable in polynomial time. Finally, it is stated in the last Section that both criteria provide a natural way for synthesizing sup-interpretations.

2 Syntax and semantics of first order programs

A program is defined formally as a quadruple $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ with $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ finite disjoint sets which represent respectively the variables, the constructor symbols and the function symbols and $\mathcal{R}$ a finite set of rules defined in Figure 1.

The set of rules induces a rewriting relation $\rightarrow$. The relation $\xrightarrow{*}$ is the reflexive and transitive closure of $\rightarrow$. Throughout the paper, we only consider programs having disjoint and linear patterns. So each program is confluent [13].

The domain of computation of a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ is the constructor algebra $\mathcal{T}(\mathcal{C})$. A ground substitution σ is a mapping from variables to values of

$$
\begin{array}{llll}
\texttt{(Values)} & \mathcal{T}(\mathcal{C}) \ni v & ::= & \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \\
\texttt{(Terms)} & \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t & ::= & \mathbf{c} \mid x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\
\texttt{(Patterns)} & \mathcal{P} \ni p & ::= & \mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n) \\
\texttt{(Rules)} & \mathcal{R} \ni r & ::= & \mathtt{f}(p_1, \cdots, p_n) \to t
\end{array}
$$

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

Fig. 1. Syntax of the programs

$\mathcal{T}(\mathcal{C})$. Given a term t and a ground substitution σ, if $t\sigma \xrightarrow{*} w$ and w is in $\mathcal{T}(\mathcal{C})$ then $[\![t\sigma]\!] = w$, $[\![t\sigma]\!] = \bot$ otherwise.

The size $|t|$ of a term t is defined to be the number of symbols of arity strictly greater than 0 occurring in t.

Example 1. Consider the following program which computes the logarithm function:

$$
\begin{array}{rl|rl}
\mathtt{log}(\mathbf{0}) & \to \mathbf{0} & \mathtt{half}(\mathbf{0}) & \to \mathbf{0} \\
\mathtt{log}(\mathbf{S}(\mathbf{0})) & \to \mathbf{0} & \mathtt{half}(\mathbf{S}(\mathbf{0})) & \to \mathbf{0} \\
\mathtt{log}(\mathbf{S}(\mathbf{S}(y))) & \to \mathbf{S}(\mathtt{log}(\mathbf{S}(\mathtt{half}(y)))) & \mathtt{half}(\mathbf{S}(\mathbf{S}(y))) & \to \mathbf{S}(\mathtt{half}(y))
\end{array}
$$

We have for every natural number n, $[\![\mathtt{log}(\mathbf{S}^n(\mathbf{0}))]\!] = \mathbf{S}^{\lfloor \log(n) \rfloor}(\mathbf{0})$ where $\mathbf{S}^{n+1}(\mathbf{0}) = \mathbf{S}(\mathbf{S}^n(\mathbf{0}))$ and $\mathbf{S}^0(\mathbf{0}) = \mathbf{0}$.

A *context* is an expression $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ with only one occurrence of each $\diamond_i$, where the $\diamond_i$ are fresh symbols. The substitution of each $\diamond_i$ by an expression d_i is noted $\mathsf{C}[d_1, \cdots, d_r]$.

Example 2. The substitution of the variable $\diamond$ by $\mathtt{half}(y)$ in the context $\mathsf{C}[\diamond] = \mathbf{S}(\mathtt{log}(\mathbf{S}(\diamond)))$ is equal to $\mathbf{S}(\mathtt{log}(\mathbf{S}(\mathtt{half}(y))))$.

Now, we define the notion of call-tree which corresponds to the tree of function calls in one execution of a program.

Definition 1. *A* state *is a tuple $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$ where $\mathtt{f}$ is a function symbol of arity n and $v_1, \ldots, v_n$ are values. Assume that $\eta_1 = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ and $\eta_2 = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ are two states. Assume also that we have $\mathtt{f}(p_1, \cdots, p_n) \to \mathsf{C}[\mathtt{g}(e_1, \cdots, e_m)] \in \mathcal{R}$ with $\mathsf{C}[\diamond]$ a context. A* transition *between two states η_1 and η_2, noted $\eta_1 \rightsquigarrow \eta_2$, is defined by:*

1. *There is a ground substitution σ such that $p_i\sigma = v_i$ for $i = 1, \ldots, n$*
2. *and $[\![e_j\sigma]\!] = u_j$ for $j = 1, \ldots, m$.*

We call such a tree a call-tree of $\mathtt{f}$ from values $u_1, \ldots, u_n$ if $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$ is its root.

A state may be seen as a stack frame since it contains a function call and its respective arguments. A call-tree of root $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$ represents all the stack frames which will be pushed on the stack when we compute $\mathtt{f}(v_1, \ldots, v_n)$. We will sometimes refer to $\overset{+}{\rightsquigarrow}$ as the transitive closure of $\rightsquigarrow$.

The size of a state is the sum of the sizes of its values. Given a call-tree and two of its states $\langle \mathtt{f}, t_1, \cdots, t_n \rangle$ and $\langle \mathtt{g}, u_1, \cdots, u_m \rangle$, such that $\langle \mathtt{f}, t_1, \cdots, t_n \rangle \overset{+}{\rightsquigarrow} \langle \mathtt{g}, u_1, \cdots, u_m \rangle$, then $\langle \mathtt{f}, t_1, \cdots, t_n \rangle \overset{+}{\rightsquigarrow} \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ is called a branch of the call-tree. The size of a branch is defined to be the sum of the sizes of all its states. The length of a branch is the number of states in the branch. Finally, the size of the call-tree is the sum of the sizes of all its states.

3 The Dependency Pair Method

In this section, we briefly review the notion of dependency pair introduced by Arts and Giesl [1] in order to analyze the termination of programs automatically.

Definition 2 (Dependency pair). *Given a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$, a dependency pair $< \mathtt{f}(p_1, \cdots, p_n), \mathtt{g}(e_1, \cdots, e_m) >$ is a couple satisfying $\mathtt{f}, \mathtt{g} \in \mathcal{F}$ and there is a context $\mathsf{C}[\diamond]$ such that $\mathtt{f}(p_1, \cdots, p_n) \rightarrow \mathsf{C}[\mathtt{g}(e_1, \cdots, e_m)] \in \mathcal{R}$.*

Example 3. If we consider the program of Example 1, $\langle \mathtt{log}(\mathbf{S}(\mathbf{S}(y))), \mathtt{half}(y) \rangle$ is a dependency pair since there is a rule $\mathtt{log}(\mathbf{S}(\mathbf{S}(y))) \rightarrow \mathsf{C}[\mathtt{half}(y)]$ with a context $\mathsf{C}[\diamond] = \mathbf{S}(\mathtt{log}(\mathbf{S}(\diamond)))$.

Definition 3 (Dependency pair graph). *We define the dependency pair graph of a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ by:*

- *The nodes are the dependency pairs*
- *Given two dependency pairs $u =< \mathtt{f}_1(p_1, \cdots, p_n), \mathtt{f}_2(e_1, \cdots, e_m) >$ and $v =< \mathtt{f}_3(q_1, \cdots, q_k), \mathtt{f}_4(d_1, \cdots, d_l) >$, there is an edge from u to v if $\mathtt{f}_2 = \mathtt{f}_3$.*

A cycle of dependency pairs is defined to be a cycle in the dependency pair graph. We say that the dependency pair u is involved in a cycle if u belongs to a cycle in the dependency graph.

The notions of dependency pairs and dependency pair graph allow to derive a termination criterion:

Theorem 1 ([1]). *A program $\mathtt{p}$ is terminating if there is a well-founded weakly monotonic quasi-ordering $\geq_{q.o.}$, closed under substitution, such that:*

1. *For each rule $\mathtt{f}(p_1, \cdots, p_n) \rightarrow e \in \mathcal{R}$, $\mathtt{f}(p_1, \cdots, p_n) \geq_{q.o.} e$.*
2. *For each dependency pair $\langle s, t \rangle$, $s \geq_{q.o.} t$*
3. *For each cycle in the dependency pair graph, there is a dependency pair $\langle s, t \rangle$ involved in the cycle such that $s >_{q.o.} t$*

4 Polynomial and additive assignments

In this section, we define polynomial assignments over real numbers. A polynomial assignment associates a polynomial function to every symbol of a program. We have chosen to study polynomials over real numbers instead of polynomials over natural numbers since the synthesis of such assignments is decidable, under some restrictions, as an application of [3]. Following the terminology of [5], we define the notion of assignment.

Definition 4 (Assignments). *An assignment of a symbol $b \in \mathcal{F} \bigcup \mathcal{C}$ of arity n is a function $(\!|b|\!) : (\mathbb{R}^+)^n \to \mathbb{R}^+$.*

We extend assignments $(\!|-|\!)$ to terms canonically. Given a term $t = b(t_1, \cdots, t_n)$ with m variables, the assignment $(\!|t|\!)$ is a function $(\mathbb{R}^+)^m \to \mathbb{R}^+$ defined by the rules:

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

where X is a fresh variable ranging over reals.

A program assignment is an assignment $(\!|-|\!)$ defined for each symbol of the program.

Definition 5 (Polynomial assignments). *Let **Max-Poly**$\{\mathbb{R}^+\}$ be the set of functions defined to be constant functions over $\mathbb{R}^+$, projections,* max, $+$, $\times$ *and closed by composition. An assignment $(\!|-|\!)$ is said to be polynomial if for each symbol $b \in \mathcal{F} \bigcup \mathcal{C}$, $(\!|b|\!)$ is a function in **Max-Poly**$\{\mathbb{R}^+\}$.*

Now we define the notion of additive assignments which guarantees that the assignment of a value remains affinely bounded by its size.

Definition 6 (Additive assignments). *An assignment of a constructor symbol $\mathbf{c}$ of arity $n > 0$ is additive if*

$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}}, \text{ with a constant } \alpha_{\mathbf{c}} \geq 1.$$

An assignment $(\!|-|\!)$ of a program $\mathtt{p}$ is additive *if each constructor symbol of $\mathtt{p}$ has an additive assignment.*

Lemma 1. *Given an additive assignment $(\!|-|\!)$, there is a constant α such that for each value v, the following inequality is satisfied:*

$$|v| \leq_\beta (\!|v|\!) \leq_\beta \alpha \times |v|$$

Proof. Define $\alpha = \max_{\mathbf{c} \in \mathcal{C}}(\alpha_{\mathbf{c}})$ where $\alpha_{\mathbf{c}}$ is taken to be the constant of definition 6, if $\mathbf{c}$ is of strictly positive arity, and $\alpha_{\mathbf{c}}$ is equal to the constant $(\!|\mathbf{c}|\!)$ otherwise. The inequalities follow directly by induction on the size of a value. □

Since the dependency pair method requires a well-founded quasi-ordering and we are working over real numbers, we first have to restrict the considered inequalities in order to preserve the well-foundedness. Following [16], we define a well-founded quasi-ordering:

Definition 7 ([16]). *Given a fixed constant $\beta > 0$, we define the quasi-ordering $\geq_\beta$ over $\mathbb{R}^+$, and its strict part, by:*

- $x \geq_\beta y$ *iff* $x \geq y$

– $x >_\beta y$ *iff for every* $x, y \in \mathbb{R}^+$, $x \geq \beta + y$

Definition 8 (Monotonic assignments). *An assignment is* monotonic *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 all* $X_1, \cdots, X_n, Y_1, \cdots, Y_n$ *of* $\mathbb{R}^+$ *with* $X_i \geq_\beta Y_i$, *we have* $(\!|b|\!)(X_1, \cdots, X_n) \geq_\beta (\!|b|\!)(Y_1, \cdots, Y_n)$.

5 A criterion for polynomial space computations

We present in this section a criterion using the dependency pair method and polynomial assignments which allows to characterize the set of functions computable in polynomial space.

Definition 9 (polynomial space interpretation). *A program* $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ *admits a polynomial space interpretation if there is an additive, monotonic and polynomial assignment* $(\!|-|\!)$ *such that its dependency pair graph satisfies:*

– *For each rule* $\mathtt{f}(p_1, \cdots, p_n) \to e \in \mathcal{R}$, $(\!|\mathtt{f}(p_1, \cdots, p_n)|\!) \geq_\beta (\!|e|\!)$.
– *For each dependency pair* $\langle s, t \rangle$, $(\!|s|\!) \geq_\beta (\!|t|\!)$
– *For each cycle in the dependency pair graph, there is a dependency pair* $\langle s, t \rangle$ *involved in the cycle such that* $(\!|s|\!) >_\beta (\!|t|\!)$

Example 4. The program of Example 1 admits the following polynomial space interpretation:

$$\begin{array}{rcl} (\!|\mathbf{0}|\!) = 0 & \Big| & (\!|\mathbf{S}|\!)(X) = X + 1 \\ (\!|\mathtt{half}|\!)(X) = X/2 & \Big| & (\!|\mathtt{log}|\!)(X) = X \end{array}$$

Indeed, for every rule, we check that:

$$\begin{aligned} (\!|\mathtt{half}(\mathbf{0})|\!) = 0 \geq_\beta 0 &= (\!|\mathbf{0}|\!) \\ (\!|\mathtt{half}(\mathbf{S}(\mathbf{0}))|\!) = 1/2 \geq_\beta 0 &= (\!|\mathbf{0}|\!) \\ (\!|\mathtt{half}(\mathbf{S}(\mathbf{S}(y)))|\!) = (Y+2)/2 \geq_\beta Y/2 + 1 &= (\!|\mathbf{S}(\mathtt{half}(y))|\!) \\ (\!|\mathtt{log}(\mathbf{0})|\!) = 0 \geq_\beta 0 &= (\!|\mathbf{0}|\!) \\ (\!|\mathtt{log}(\mathbf{S}(\mathbf{0}))|\!) = 1 \geq_\beta 0 &= (\!|\mathbf{0}|\!) \\ (\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!) = Y + 2 \geq_\beta 2 + Y/2 &= (\!|\mathbf{S}(\mathtt{log}(\mathbf{S}(\mathtt{half}(y))))|\!) \end{aligned}$$

Moreover, taking $\beta = 1$, for every dependency pair, we have:

$$\begin{aligned} (\!|\mathtt{half}(\mathbf{S}(\mathbf{S}(y)))|\!) = Y/2 + 1 &>_\beta Y/2 = (\!|\mathtt{half}(y)|\!) \\ (\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!) = Y + 2 &>_\beta 1 + Y/2 = (\!|\mathtt{log}(\mathbf{S}(\mathtt{half}(y)))|\!) \\ (\!|\mathtt{log}(\mathbf{S}(\mathbf{S}(y)))|\!) = Y + 2 &\geq_\beta Y/2 = (\!|\mathtt{half}(y)|\!) \end{aligned}$$

Just notice that the inequalities corresponding to the two cycles in the dependency pair graph are strict.

First, we are going to show that a program whose dependency pair graph admits a polynomial space interpretation is terminating:

Theorem 2. *Suppose that $\mathtt{p}$ is program whose dependency pair graph admits a polynomial space interpretation $(\!|-|\!)$, then the program $\mathtt{p}$ terminates on all inputs.*

Proof. Just define the quasi-ordering $\geq_{q.o.}$ on terms to be $\geq_\beta$. By definition of a polynomial space interpretation the criterion of the dependency pair method is checked. It remains to see that this quasi-ordering is monotonic well-founded and closed by substitution. It is well-founded since the strict inequality $>_\beta$ guarantees a decrease by at least the fixed constant $\beta > 0$. It is closed under substitution by definition of assignments. Finally, it is monotonic since assignments are monotonic. □

Proposition 1. *Given a program $\mathtt{p}$ which admits a polynomial space interpretation $(\!|-|\!)$, then for every term t and every ground substitution σ such that $t\sigma \to^* u$, we have :*

$$(\!|t\sigma|\!) \geq_\beta (\!|u|\!)$$

Proof. We show this result by induction on the derivation length. Consider a term $e = b(e_1, \cdots, e_n)$ and suppose that $b(e_1, \cdots, e_n)\sigma \to^* u$. By induction hypothesis (I.H.), and using a call-by-value evaluation, if $e_i\sigma \to^* u_i$ then $(\!|e_i\sigma|\!) \geq_\beta (\!|u_i|\!)$. We can evaluate programs in such a way since they are confluent. Moreover, suppose that $b(u_1, \cdots, u_n) \to^* u$ then, by induction hypothesis again, we have $(\!|b(u_1, \cdots, u_n)|\!) \geq_\beta (\!|u|\!)$. So, that :

$$\begin{array}{lll}
(\!|e\sigma|\!) =_\beta (\!|b(e_1, \cdots, e_n)\sigma|\!) & & \text{Since } e = b(e_1, \cdots, e_n) \\
=_\beta (\!|b|\!)((\!|e_1\sigma|\!), \ldots, (\!|e_n\sigma|\!)) & & \text{By Definition of } (\!|-|\!) \\
\geq_\beta (\!|b|\!)((\!|u_1|\!), \ldots, (\!|u_n|\!)) & & \text{By I.H. and Monotonicity} \\
=_\beta (\!|b(u_1, \cdots, u_n)|\!) & & \text{By Definition of } (\!|-|\!) \\
\geq_\beta (\!|u|\!) & & \text{By I.H. again}
\end{array}$$

□

Theorem 3. *Given a program $\mathtt{p}$ admitting a polynomial space interpretation, then for every function symbol $\mathtt{f}$ of arity n there is a polynomial such that for every values $v_1, \cdots, v_n$ we have:*

$$P(|v_1|, \cdots, |v_n|) \geq |[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|$$

Proof. By Theorem 2, the program terminates so that $[\![\mathtt{f}(v_1, \cdots, v_n)]\!]$ is clearly defined. By Proposition 1, we have $(\!|\mathtt{f}(v_1, \cdots, v_n)|\!) \geq_\beta (\!|[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|\!)$. We set $P(\diamond_1, \cdots, \diamond_n) = (\!|\mathtt{f}|\!)(\alpha \times \diamond_1, \ldots, \alpha \times \diamond_n)$ with α the constant of Lemma 1. We obtain that $P(|v_1|, \ldots, |v_1|) \geq |[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|$ by monotonicity of assignments. □

Lemma 2. *Given a program $\mathtt{p}$ admitting a polynomial space interpretation $(\!|-|\!)$ and a call-tree corresponding to one execution of this program and containing a branch of the shape $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{+}{\rightsquigarrow} \langle \mathtt{f}, u_1, \cdots, u_n\rangle$ then:*

$$(\!|\mathtt{f}(v_1, \cdots, v_n)|\!) >_\beta (\!|\mathtt{f}(u_1, \cdots, u_n)|\!)$$

Proof. This result is obtained by combining Definition 9, which guarantees that each cycle of a dependency pair graph corresponds to a strict decrease of the polynomial space interpretation, Proposition 1 and the monotonicity of assignments. □

Corollary 1. *Given a program $\mathtt{p}$ admitting a polynomial space interpretation $(\!|-|\!)$ then every branch of the call-tree of the shape $\langle \mathtt{f}, v_1, \cdots, v_n\rangle \overset{+}{\rightsquigarrow} \langle \mathtt{f}, u_1, \cdots, u_n\rangle$ has a length bounded by $\gamma \times (\!|\mathtt{f}(v_1, \cdots, v_n)|\!)$, for some constant γ.*

Proof. By Lemma 2, we know that two successive calls of the same function symbol corresponds to a strict decrease $(\!|f(v_1, \cdots, v_n)|\!) >_\beta (\!|f(u_1, \cdots, u_n)|\!)$. It means that $(\!|f(v_1, \cdots, v_n)|\!) \geq \beta + (\!|f(u_1, \cdots, u_n)|\!)$. Consequently, we have at most $(\!|f(v_1, \cdots, v_n)|\!)/\beta$ successive states in the branch where the function symbol $\mathtt{f}$ occurs. Since the maximal number of dependency pairs involved in a cycle of the dependency pair graph is bounded by the size of the program, which is fixed, we obtain that the branch corresponding to the cycle has a length bounded by $\gamma \times (\!|f(v_1, \cdots, v_n)|\!)$, for some constant γ. □

Lemma 3. *Given a program $\mathtt{p}$ admitting a polynomial space interpretation $(\!|-|\!)$ and a call-tree corresponding to one execution of this program, then the size of each branch of the call-tree is polynomially bounded in the size of the inputs.*

Proof. By Theorem 3, we know that every value of a state is polynomially bounded by the size of the inputs. That is, there is a polynomial R such that for every state $\langle \mathtt{g}, v_1, \cdots, v_m\rangle$ of a call-tree of root $\langle f, u_1, \cdots, u_n\rangle$, we have:

$$\forall i \in \{1, m\}\,,\ |v_i| \leq R(\max_{j=1..n}(|u_j|))$$

So that, the size of each state is bounded by $Q(\max_{j=1..n}|u_j|)$ with $Q(X) = k \times R(X)$ and k the maximal arity of the program. By Corollary 1, we have shown that each cycle starting from $\langle \mathtt{g}, v_1, \cdots, v_m\rangle$ has at most $\gamma \times (\!|\mathtt{g}(v_1, \cdots, v_m)|\!)$ occurrences, which is bounded by $P_{\mathtt{g}}(|v_1|, \cdots, |v_n|) = \gamma \times (\!|\mathtt{g}|\!)(\alpha \times |v_1|, \ldots, \alpha \times |v_k|)$ with α the constant of Lemma 1. Consequently, each cycle starting from $\langle \mathtt{g}, v_1, \cdots, v_m\rangle$ has at most $P_{\mathtt{g}}(Q(\max_{j=1..n}|u_j|), \ldots, Q(\max_{j=1..n}|u_j|))$ occurrences. Now define $\omega(X) = \max_{\mathtt{g}\in\mathcal{F}}(P_{\mathtt{g}}(Q(X), \ldots, Q(X)))$. Let A be the maximal number of cycles in the program (Notice that A is considered as a constant since it only depends on the size of the program). We know that the depth of each branch starting from $\langle f, u_1, \cdots, u_n\rangle$ is bounded by $A \times \omega(\max_{j=1..n}(|u_j|))$. Finally, $A \times \omega(\max_{j=1..n}(|u_j|)) \times Q(\max_{j=1..n}(|u_j|))$ is the required polynomial bound on the size of each branch. □

Theorem 4. *The set of functions computed by programs which admits a polynomial space interpretation is exactly the set of functions computable in polynomial space.*

Proof. By Lemma 3, we know that the size of each branch and each state of the call-tree is polynomially bounded in the size of the inputs. Evaluating the program in the depth of the call-tree, we obtain that the set of functions computed by programs which admit a polynomial space interpretation is included in PSPACE. The proof of completeness is inspired by a characterization of [5] using Parallel Register Machines (PRM). Savitch [20] and Chandra, Kozen and Stockmeyer [6] have shown that the set of functions computed by PRM in polynomial time is exactly the set of functions computable in polynomial space. We let the reader check that the program given in [5], which simulates PRM by a TRS admits clearly a polynomial space interpretation. □

This criterion is an improvement of the previous characterization of [5] using the notion of quasi-interpretation and Recursive Path Orderings (RPO) with lexicographic and product status. Indeed, our criterion has a greater intensionality since it captures more natural algorithms, like the ones computing the greatest common divisor or the division of two unary numbers, as illustrated in the following example.

Example 5. Consider the following program which computes the greatest common divisor:

$$\begin{array}{rl|rl}
\mathbf{le}(\mathbf{S}(x),\mathbf{S}(y)) & \to \mathbf{le}(x,y) & \mathtt{minus}(\mathbf{0},z) & \to \mathbf{0}\\
\mathbf{le}(\mathbf{S}(x),\mathbf{0}) & \to \mathbf{tt} & \mathtt{minus}(\mathbf{S}(z),\mathbf{0}) & \to \mathbf{S}(z)\\
\mathbf{le}(\mathbf{0},\mathbf{S}(y)) & \to \mathbf{ff} & \mathtt{minus}(\mathbf{S}(u),\mathbf{S}(v)) & \to \mathtt{minus}(u,v)\\
\mathtt{if}(\mathbf{tt},u,v) & \to u & \mathtt{if}(\mathbf{ff},u,v) & \to v\\
\mathtt{gcd}(\mathbf{0},z) & \to z & \mathtt{gcd}(\mathbf{S}(z),\mathbf{0}) & \to \mathbf{S}(z)
\end{array}$$

$$\mathtt{gcd}(\mathbf{S}(u),\mathbf{S}(v)) \to \mathtt{if}(\mathbf{le}(u,v),\mathtt{gcd}(\mathtt{minus}(v,u),\mathbf{S}(u)),\mathtt{gcd}(\mathtt{minus}(u,v),\mathbf{S}(v)))$$

This program is not terminating by RPO since the last rule calls the function symbol `minus` on the arguments of a recursive call. However, we let the reader checks that it admits the following polynomial space interpretation:

$$\begin{array}{rl|rl}
(\!|\mathbf{0}|\!) & = (\!|\mathbf{tt}|\!) = (\!|\mathbf{ff}|\!) = 0 & (\!|\mathbf{S}|\!)(X) & = X+1\\
(\!|\mathbf{le}|\!)(X,Y) & = (\!|\mathtt{minus}|\!)(X,Y) = X & (\!|\mathtt{if}|\!)(X,Y,Z) & = \max(Y,Z)\\
(\!|\mathtt{gcd}|\!)(X,Y) & = X+Y &&
\end{array}$$

6 A criterion for polynomial time computations

We present in this section a refinement of the previous criterion which allows to apply the dependency pair method using polynomial assignments in order to capture polynomial time computable functions. First, we define the notion of agglomerated set which allows to control the recursive calls corresponding to the same recursive rule together:

Definition 10 (Precedence). *We define a precedence $\geq_{\mathcal{F}}$ on function symbols. Set $\mathtt{f} \geq_{\mathcal{F}} \mathtt{g}$ if there is a rule of the shape $\mathtt{f}(\overline{p}) \to \mathsf{C}[\mathtt{g}(\overline{e})] \in \mathcal{R}$. Then, take the reflexive and transitive closure of $\geq_{\mathcal{F}}$, that we also note $\geq_{\mathcal{F}}$. $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ if $\mathtt{f} \geq_{\mathcal{F}} \mathtt{g}$ and inversely $\mathtt{g} \geq_{\mathcal{F}} \mathtt{f}$. Lastly, $\mathtt{f} >_{\mathcal{F}} \mathtt{g}$ if $\mathtt{f} \geq_{\mathcal{F}} \mathtt{g}$ and $\mathtt{g} \geq_{\mathcal{F}} \mathtt{f}$ does not hold. Intuitively, $\mathtt{f} \geq_{\mathcal{F}} \mathtt{g}$ means that $\mathtt{f}$ calls $\mathtt{g}$ in some executions. And $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ means that $\mathtt{f}$ and $\mathtt{g}$ call each other recursively.*

Definition 11 (Agglomerated set). *Given a dependency pair $< s, t >$ of the dependency pair graph, we define its agglomerated set $A(< s, t >)$ by:*

$$A(< s, t >) = \{< s', t' > \text{ such that } s = s' = \mathtt{f}(p_1, \cdots, p_n),\ t' = \mathtt{g}(e_1, \cdots, e_m) \text{ and } \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \ \}$$

Definition 12 (polynomial time interpretation). *A program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ admits a polynomial time interpretation if there is an additive, monotonic and polynomial assignment $(\!|-|\!)$ such that its dependency pair graph satisfies:*

- *For each rule $\mathtt{f}(p_1, \cdots, p_n) \to e \in \mathcal{R}$, $(\!|\mathtt{f}(p_1, \cdots, p_n)|\!) \geq_\beta (\!|e|\!)$.*
- *For each dependency pair $\langle s, t \rangle$, $(\!|s|\!) \geq_\beta (\!|t|\!)$*
- *For each agglomerated set $A(< s, t >) = \{< s, t_1 >, \ldots, < s, t_n >\}$, we have $(\!|s|\!) >_\beta \sum_{i=1}^{n} (\!|t_i|\!)$*

Remark 1. Notice that the last condition corresponds to a resource sharing between the distinct recursive calls. It is very close in the spirit from the resource diamond introduced by Hoffman [12] and from Soft Linear Logic [14]. Moreover, this condition also implies that for every cycle of the dependency pair graph, there is a dependency pair $< s, t >$ such that $(\!|s|\!) >_\beta (\!|t|\!)$ since $(\!|s|\!) >_\beta \sum_{i=1}^{n} (\!|t_i|\!)$ implies $\forall j \in \{1, n\}\ (\!|s|\!) >_\beta (\!|t_j|\!)$ and every polynomial time interpretation is a polynomial space interpretation. Consequently, Proposition 1 and Theorems 3 and 2 hold for polynomial time interpretations.

Example 6. Just check that the program of Example 1 admits the following polynomial time interpretation:

$$\begin{array}{rcl} (\!|\mathbf{0}|\!) = 0 & \quad\vert\quad & (\!|\mathbf{S}|\!)(X) = X + 1 \\ (\!|\mathtt{half}|\!)(X) = X & & (\!|\mathtt{log}|\!)(X) = X \end{array}$$

Lemma 4. *Given a program $\mathtt{p}$ admitting a polynomial time interpretation $(\!|-|\!)$ and a call-tree corresponding to one execution of this program, then the size of the call-tree is polynomially bounded in the size of the inputs.*

Proof. By Theorem 3, we know that every value of a state is polynomially bounded by the size of the inputs. That is, there is a polynomial R such that for every state $\langle \mathtt{g}, v_1, \cdots, v_m \rangle$ of a call-tree of root $\langle \mathtt{f}, u_1, \cdots, u_n \rangle$, we have:

$$\forall i \in \{1, m\}\,,\ |v_i| \leq R(\max_{j=1..n}(|u_j|))$$

If the state $\langle \mathtt{g}, v_1, \cdots, v_m \rangle$ corresponds to a recursive call, then suppose that its corresponding agglomerated set is of the shape

$$\{< \mathtt{g}(p_1, \cdots, p_n), \mathtt{f}_1(e_1) >, \ldots, < \mathtt{g}(p_1, \cdots, p_n), \mathtt{f}_n(e_n) >\}$$

There is a ground substitution σ such that $p_i\sigma = v_i$. The third condition of the polynomial time interpretation implies that:

$$\begin{aligned} (\!|\mathtt{g}(v_1, \cdots, v_n)|\!) &>_\beta \sum_{i=1}^{n} (\!|\mathtt{f}_i(e_i\sigma)|\!) && \text{By Definition 12} \\ &\geq \beta + \sum_{i=1}^{n} (\!|\mathtt{f}_i([\![e_i\sigma]\!])|\!) && \text{By Proposition 1} \end{aligned}$$

Consequently, we can apply at most $1/\beta \times (\!|\mathtt{g}(v_1, \cdots, v_n)|\!)$ recursive rules. And the number of states added in the call-tree by the rules corresponding to a cycle in the dependency pair graph is polynomially bounded by the inputs size. Since the number of cycles in the dependency pair graph is bounded by the size of the program, we obtain that the number of states in the call-tree is polynomially bounded by the inputs size. □

Theorem 5. *The set of functions computed by programs which admits a polynomial time interpretation is exactly the set of functions computable in polynomial time.*

Proof. By Lemma 4, we know that the size of the call-tree is polynomially bounded by the input sizes, and consequently the derivation length is polynomially bounded. Theorem 3 states that the size of any computed value is polynomially bounded by the input sizes. As a consequence, we can apply the proof of the characterization of the polynomial time computable functions with polynomial interpretations of [2], showing that every program can be simulated by a Turing Machine in polynomial time.

Conversely, every polynomial interpretation of [2] is a polynomial time interpretation. Indeed, the subterm property and the definition of polynomial interpretations in [2] entail all the conditions of Definition 12. □

This criterion is an improvement of the characterization of [2] for two reasons. Firstly, we work on real numbers so that we have more flexibility for the choice of the assignments. Moreover, because of Tarski's result assignments of bounded degree over real numbers are decidable so that we have procedure in order to find an assignment which is a polynomial time interpretation. Secondly, we only require a strict inequality for the function calls which correspond to cycles in the dependency pair graph. As a consequence, this criterion has less constraints and captures more algorithms. It also improves the characterization of polynomial time computable functions using Recursive Path Orderings (RPO) with product status and quasi-interpretations in [5]. Indeed, our criterion has a greater intensionality since it captures more natural algorithms, like the one computing the division of two numbers, as illustrated by the following example.

Example 7. Consider the following program which computes the greatest common divisor:

$$\begin{array}{rcl} \mathtt{minus}(\mathbf{0}, z) & \to & \mathbf{0} \\ \mathtt{minus}(\mathbf{S}(z), \mathbf{0}) & \to & \mathbf{S}(z) \\ \mathtt{minus}(\mathbf{S}(u), \mathbf{S}(v)) & \to & \mathtt{minus}(u, v) \end{array} \qquad \begin{array}{rcl} \mathtt{quo}(\mathbf{0}, \mathbf{S}(z)) & \to & \mathbf{0} \\ \mathtt{quo}(\mathbf{S}(y), \mathbf{S}(z)) & \to & \mathbf{S}(\mathtt{quo}(\mathtt{minus}(y, z), \mathbf{S}(z))) \end{array}$$

This program is not terminating by RPO with product status since the last rule of quo calls the function minus on the arguments of a recursive call. However, we let the reader checks that it admits the following polynomial time interpretation:

$$\begin{array}{c|c} (\!|\mathbf{0}|\!) = 0 & (\!|\mathtt{minus}|\!)(X,Y) = X \\ (\!|\mathbf{S}|\!)(X) = X+1 & (\!|\mathtt{quo}|\!)(X,Y) = X+Y \end{array}$$

7 Sup-interpretation synthesis

A sup-interpretation is a tool [18, 19] which provides an upper-bound on the size of a value computed by a symbol. In [4], we show that sup-interpretations allow to characterize the set of alternating logarithmic time computable functions. We show in this section that polynomial time and space assignments give heuristics to compute polynomial sup-interpretations.

Definition 13 (Sup-interpretation). *A sup-interpretation is a partial assignment θ which verifies the three conditions below:*

1. *The assignment θ is weakly monotonic. That is, for each symbol b in the domain of θ, the function $\theta(b)$ is a monotonic function.*
2. *For each value v of the computational domain $\mathcal{T}(\mathcal{C})$, the sup-interpretation of v is greater than the size of v, i.e. $\theta(v) \geq |v|$.*
3. *For each symbol b in the domain of θ of arity n and for each values $v_1, \ldots, v_n$ of $\mathcal{T}(\mathcal{C})$, if $[\![b(v_1, \ldots, v_n)]\!] \in \mathcal{T}(\mathcal{C})$, then $\theta(b(v_1, \ldots, v_n)) \geq \theta([\![b(v_1, \ldots, v_n)]\!])$.*

Theorem 6. *Every polynomial time or polynomial space interpretation is a sup-interpretation.*

Proof. By Proposition 1, for every polynomial time or polynomial space interpretation we have $(\!|\mathtt{f}(v_1, \cdots, v_n)|\!) \geq (\!|[\![\mathtt{f}(v_1, \cdots, v_n)]\!]|\!)$. Since every polynomial time or polynomial space interpretation is additive, by Lemma 1, we have for every value v, $(\!|v|\!) \geq |v|$. Finally, it remains to see that polynomial time or polynomial space interpretation consist in monotonic assignments. □

A consequence of this Theorem is that we have a procedure to compute sup-interpretations. Indeed, if we manage to compute a polynomial time or polynomial space assignment then we obtain a sup-interpretation.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.
3. G. Bonfante, J.-Y. Marion, J.-Y. Moyen, and R. Péchoux. Synthesis of quasi-interpretations. *LCC2005, LICS affiliated Workshop*, 2005. `http://hal.inria.fr/`.

4. G. Bonfante, J.-Y. Marion, and R. Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR*, volume 4246 of *Lecture Notes in Artificial Intelligence*, pages 90–104, 2006.
5. G. Bonfante, J.Y. Marion, and J.Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2007.
6. A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.
7. E.A Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE*, number 607 in Lecture Notes in Artificial Intelligence, pages 139–147, 1992.
8. E. Contejean, C. Marche, B. Monate, and X. Urbain. Proving Termination of Rewriting with CiME. *WST*, pages 71–73, 2003.
9. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.
10. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. *IJCAR*, 4130:281–286.
11. N. Hirokawa and A. Middeldorp. Tyrolean termination tool. *Technical Report AIB-2004-07, RWTH*, pages 59–62, 2004.
12. M. Hofmann. The strength of Non-Size Increasing computation. In *POPL*, pages 260–269, 2002.
13. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
14. Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318:163–180, 2004.
15. D.S. Lankford. On proving term rewriting systems are noetherien. Technical report, 1979.
16. S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
17. Z. Manna and S. Ness. On the termination of Markov algorithms. In *Third hawaii international conference on system science*, pages 789–792, 1970.
18. J.-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, volume 3945 of *LNCS*, pages 163–176, 2006.
19. J.Y. Marion and R. Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Transactions on Computational Logic*. Submitted.
20. W. J. Savitch. Relationship between nondeterministic and deterministic tape classes. *Journal of Computer System Science*, 4:177–192, 1970.