On complexity analysis by Quasi-interpretation

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

Abstract

We present a survey on a method to analyse the program time and space complexity, based on termination orderings and quasi-interpretation. This method can be implemented to certify the runtime (or space) of programs. We demonstrate that the class of functions computed by first order functional programs over free algebras which terminate by permutation Path Ordering (resp. Lexicographic Path Ordering) and admit a quasi-interpretation bounded by a polynomial, is exactly the class of functions computable in polynomial time (resp. space).

1 Introduction

This paper is part of a general investigation on program complexity analysis. We have introduced program quasi-interpretations, see [24, 25, 26, 7], in order to bound the size of program inputs and outputs. Combined with termination methods, we can determine the runtime (or space) of a program.

In this survey, we consider first order functional programs for which termination is established by using recursive path orderings. Suppose that a programs admits a quasi-interpretation which is bounded by a polynomial. We obtain different resource upper bounds base on termination proofs.

- If the program terminates by permutation path ordering, then it is computable in polynomial time. It is worth noticing that we have to compute the program by call-by value semantics with a cache.
- If the program terminates by lexicographic path ordering, then the computation consumes a polynomial space.

From a practical point of view, the bottom line is this. The complexity analysis can be performed by static analysis and can be partially automatized. Moreover, we can build resource certificates similar to the idea behind proof-carrying code techniques. Indeed, Krishnamoorthy and Narendran in [20] have proved that termination by recursive path orderings is NP-complete. To find a quasi-interpretation is not too difficult in general, because the program denotation turns out to be a good candidate, see [27]. Moreover, Amadio shows in [1] that synthesis max-plus quasi-interpretations is NP-hard. These ideas have been applied in a resource byte-code verifier [2].

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

From a theoretical point of view, we characterize the class PTIME and PSPACE. This work is related to Bellantoni and Cook [4], Leivant [22] and Marion [23] ideas to delineate complexity classes. However most of the complexity class characterizations focus on functions and not on algorithms. A consequence of this kind of extensional approaches is the fact they fail to say something meaningful about programming theory. For this reason, we are more interested by the algorithms than by functions. Hence, we move from extensional characterizations to intensional ones. Such intensional approach have actually been studied by Caseiro [8] and Hofmann [18].

2 First order functional programming

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

2.1 Syntax of programs

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

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

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

Definition 2. A program is a quintuplet $\mathtt{main} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \mathtt{f} \rangle$ such that:

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

All along the paper, we assume that the set of rules is implicit and we use `main` to denote the program and its main symbol.

2.2 Semantics

The semantics is given by the term rewriting system $\mathcal{E}$. We recall briefly some vocabulary of rewriting theories. For further details, one might consult Dershowitz and Jouannaud's survey [14] from which we take the notations. The rewriting relation $\to$ induced by a program `main` is defined as follows $t \to s$ if s is obtained from t by applying one of the rules of $\mathcal{E}$. The relation $\xrightarrow{*}$ is the reflexive-transitive closure of $\to$. Lastly, $t \xrightarrow{!} s$ means that $t \xrightarrow{*} s$ and s is in

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

normal form, *i.e.* no other rule may be applied. A ground (resp. constructor) substitution is a substitution from $\mathcal{X}$ to $\mathcal{T}(\mathcal{C},\mathcal{F})$ (resp. $\mathcal{T}(\mathcal{C})$).

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

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

3 Orderings on terms

The use of orderings has been widely studied in order to prove the termination of term rewriting systems. Among them, we are interested here by path orderings and especially the *Multiset Path Ordering (MPO)* and the *Lexicographic Path Ordering (LPO)* respectively introduced by Plaisted [28] and Dershowitz [13] and by Kamin and Lévy [19]. We briefly describe them, together with some basic properties we shall use later on.

3.1 Path orderings

Definition 4. Let $M = \{m_1, \cdots, m_p\}$ and $N = \{n_1, \cdots, n_p\}$ be two multisets with the same number of elements and $\prec$ be an ordering over these elements. The permutation extension of $\prec$ over multisets is defined in the following way: $M \prec^m N$ if and only if there exists a permutation π such that

- $\forall 1 \leq i \leq p, m_i \preceq n_{\pi(i)}$.
- $\exists 1 \leq j \leq n$ such that $m_j \prec n_{\pi(j)}$.

Definition 5. Let $\prec$ be a term ordering. We note $\prec^l$ its usual lexicographic extension.

A precedence $\preceq_{\mathcal{F}}$ (strict precedence $\prec_{\mathcal{F}}$) is an ordering (strict ordering) on the set $\mathcal{F}$ of function symbols. Define the equivalence relation $\approx_{\mathcal{F}}$ as $\texttt{f} \approx_{\mathcal{F}} \texttt{g}$ iff $\texttt{f} \preceq_{\mathcal{F}} \texttt{g}$ and $\texttt{g} \preceq_{\mathcal{F}} \texttt{f}$. This order $\preceq_{\mathcal{F}}$ on function symbol extends canonically on $\mathcal{C} \cup \mathcal{F}$ by saying that

- constructors are smaller than function symbols,
- two constructors are incomparable.

It is worth noting that constructors are minimal symbols with respect to $\preceq_{\mathcal{F}}$. Moreover, we will consider that constructors and all variables are incomparable.

Definition 6. Given a precedence $\preceq_{\mathcal{F}}$, we define the Multiset Path Ordering and the Lexicographic Path Ordering by the following (inductive) rules. x should be taken in $\{m, l\}$.

$$\frac{\{s_1,\cdots,s_n\} \prec^m \{t_1,\cdots,t_n\}}{\mathbf{c}(s_1,\cdots,s_n) \prec_{xpo} \mathbf{c}(t_1,\cdots,t_n)}\,\mathbf{c}\in\mathcal{C} \qquad \frac{s \preceq_{xpo} t_i}{s \prec_{xpo} f(\ldots,t_i,\ldots)}\, f\in\mathcal{F}\bigcup\mathcal{C}$$

$$\frac{s_i \prec_{xpo} \mathtt{f}(t_1,\cdots,t_n) \qquad g \prec_{\mathcal{F}} \mathtt{f}}{g(s_1,\cdots,s_m) \prec_{xpo} \mathtt{f}(t_1,\cdots,t_n)}\, g\in\mathcal{F}\bigcup\mathcal{C}$$

$$\frac{(s_1,\cdots,s_n) \prec^x_{xpo} (t_1,\cdots,t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad s_j \prec_{xpo} \mathtt{f}(t_1,\cdots,t_n)}{\mathtt{g}(s_1,\cdots,s_n) \prec_{xpo} \mathtt{f}(t_1,\cdots,t_n)}$$

Definition 7. A program is terminating by MPO (resp. LPO) if there is a precedence on $\mathcal{F}$ such that for each rule $l \to r$ of $\mathcal{E}$, we have $r \prec_{mpo} l$ (resp. $\prec_{lpo}$).

Remark 8. Note that these orderings are special cases of the more general Recursive Path Ordering (RPO) with status *à la* Kamin & Lévy [19] where the status of each constructor is permutation and the status of functions symbols is either permutation (MPO) or lexicographic (LPO).

Note also that they doesn't exactly define the usual MPO or LPO but can easily be shown to capture the same functions (but not necessarily the same algorithms). Especially, the permutation ordering we use for MPO is a restriction of the multiset ordering usually used, but the resulting ordering is nonetheless as powerful as the original one (see [26]).

Theorem 9 (Hofbauer [17], Cichon [10]). *The set of functions computed by programs that terminate with respect to MPO is exactly the set of primitive recursive functions.*

Theorem 10 (Weiermann [30]). *The set of functions computed by programs that terminate with respect to LPO is exactly the set of multiply-recursive functions.*

3.2 Polynomial quasi-interpretation

Quasi-interpretations have been introduced by Bonfante [5] and, independently, by Marion and Moyen [26].

Definition 11 (Quasi-interpretations). A *polynomial quasi-interpretation (or shorter in this paper,* quasi-interpretation) of a symbol $a \in \mathcal{F}\bigcup\mathcal{C}$ whose arity is n is a function $(\!|a|\!) : \mathbf{R^+}^n \to \mathbf{R}^+$ such that :

- $(\!|a|\!)$ is bounded by a polynomial.
- $(\!|a|\!)(X_1,\cdots,X_n) \geq X_i$ for all $1 \leq i \leq n$.
- $(\!|a|\!)$ is increasing (not necessarily strictly) with respect to each variable.

As a consequence, the quasi-interpretation of a constant is a positive number.

We extend a quasi-interpretation $(\!|-|\!)$ to terms canonically: $(\!|f(t_1,\cdots,t_n)|\!) = (\!|f|\!)((\!|t_1|\!),\cdots,(\!|t_n|\!))$ where we take the identity for variables.

Moreover, quasi-interpretation are classified according to the quasi-interpretation of constructors. A quasi-interpretation is

- of **kind 0** if $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + b, b \geq 1$;
- of **kind 1** if $(\!|\mathbf{c}|\!)$ is a polynomial whose degree is at most 1 in each variable;
- of **kind 2** if $(\!|\mathbf{c}|\!)$ is any polynomial.

If nothing is specified, we will always consider quasi-interpretation to be of kind 0.

Definition 12. $(\!|-|\!)$ is a quasi-interpretation of a program `main` if for each rule $l \to r \in \mathcal{E}(\texttt{main})$ and for each closed substitution σ, $(\!|l\sigma|\!) \geq (\!|r\sigma|\!)$.

Remark 13. Quasi-interpretations do not ensure termination. Indeed, the rule $\texttt{f}(x) \to \texttt{f}(x)$ admits a quasi-interpretation but doesn't terminate.

Moreover, quasi-interpretation do not give enough information to decide termination as stated in the following theorem.

Theorem 14. *Let `main` be a system admitting a quasi-interpretation (any kind). Whether the system terminates or not is undecidable.*

Proof. Senizergues proved in [29] that the termination of non-increasing semi-Thue systems is undecidable. But, these semi-Thue systems are a particular case of rewriting systems with a quasi-interpretation (simply take the identity polynomial for the unary symbols and 1 for the unique constant ϵ) . The theorem follows immediately. □

But the quasi-interpretation give nonetheless some information about the complexity of the program. Indeed, a program admitting a quasi-interpretation (any kind) either terminates in triple exponential time or will loop forever, thus leading to a potential runtime detection of non-termination.

3.3 Interpretations

Definition 15. An interpretation is a quasi-interpretation with the following restrictions:

(i) $(\!|f|\!)$ is a polynomial,

(ii) $(\!|f|\!)(X_1, \cdots, X_n) > X_i$,

(iii) $(\!|l\sigma|\!) > (\!|r\sigma|\!)$.

Programs admitting an interpretation terminate. This sort of termination proof, by polynomial interpretations, was introduced by Lankford [21] and turns out to be a useful tool for proving termination (see [9, 15] among other).

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

Theorem 16 (Bonfante, Cichon, Marion and Touzet [6]). *According to its kind, interpretation characterise the following complexity classes:*

kind of the interpretation	*confluent system*	*non-confluent system*
0	PTIME	NPTIME
1	ETIME	NETIME
2	E_2TIME	NE_2TIME

Where non-confluent system are defined following Gurevich and Grädel [16]:

Definition 17 (Non confluent programs). Given a quintuplet $\mathtt{main} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}, \mathtt{f}\rangle$ and an order $\prec$ on constructor terms—actually, this order should be computable in polynomial time—, the semantics of main, that is $[\![\mathtt{main}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ where n is the arity of main is defined as follows. For all $u_i \in \mathcal{T}(\mathcal{C})$,

$$[\![\mathtt{main}]\!](u_1, \cdots, u_n) = \max_{\prec}\{v \in \mathcal{T}(\mathcal{C}) \mid \mathtt{main}(u_1, \cdots, u_n) \xrightarrow{!} v\}.$$

4 Termination orderings and quasi-interpretations

4.1 MPO and quasi-interpretations

Definition 18. A MPO^{QI}-program (of kind k) is a MPO-program that admits a quasi-interpretation (of kind k).

Theorem 19 (Marion & Moyen [26]). *The set of functions computed by MPO^{QI}-programs is exactly the set of functions computable in*

- *polynomial time for kind* 0*;*
- *exponential time for kind* 1*;*
- *double exponential time for kind* 2*.*

Example 20. Given a list l, $\mathtt{sort}(l)$ sorts the elements of l. The algorithm is the insertion sort. Constructors are $\mathcal{C} = \{\mathbf{tt}, \mathbf{ff}, \mathbf{0}, \mathbf{S}, \mathbf{nil}, \mathbf{cons}\}$.

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

This program terminates by MPO with the precedence $\mathtt{if} \prec_{\mathcal{F}} \_ < \_ \prec_{\mathcal{F}} \mathtt{insert} \prec_{\mathcal{F}} \mathtt{sort}$. It admits the following quasi-interpretation (of kind 0):

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

So, it computes a function of Ptime.

Example 21. The following algorithm computes the length of the longest common subsequence of two sequences.

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

It terminates by MPO with the precedence $\mathtt{max} \prec_{\mathcal{F}} \mathtt{lcs}$ and it admits the following quasi-interpretation (of kind 0):

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

So, it computes a function of PTIME.

This latest example is particularly interesting. Indeed, if one apply the rules of the program, one may get a exponentially long derivation chain, but the theorem states that it can be computed in polynomial time. Actually, one should be careful not to confuse the algorithm and the function it computes. This function (longest common subsequence) is a classical textbook example of so called "dynamic programming" (see chapter 16 of [12]) and can in this way be computed in polynomial time.

So, the theorem doesn't characterise 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 call its *implicit* complexity.

Of course, one may ask whether the polynomial bound is achievable or not (we mean automatically achievable from the program). And it actually is, simply by simulating at runtime the dynamic programming technique, that is storing each and every result of a function call in a table and avoiding to recompute the same function call if it's already in the table. This technique is inspired from Jones' rereading of Cook classical technique over 2 way push-down automatas ([11, 3]) and is called memoisation.

This memoisation-evaluation of the program is implemented by the cache-interpreter of Figure 1. In the general case, memoisation is not used because one cannot decide which result will be reused and the cache may become too big to be really useful. In our particular case, the termination ordering gives enough informations on the structure of the program to allow minimisation of the cache [24].

4.2 LPO and quasi-interpretations

Definition 22. A LPO^{QI}-program (of kind k) is a LPO-program that admits a quasi-interpretation (of kind k).

Theorem 23 (Bonfante, Marion & Moyen [7]). *The set of functions computed by LPO^{QI}-programs is exactly the set of functions computable in*

- *polynomial space for kind* 0*;*
- *exponential space for kind* 1*;*
- *double exponential space for kind* 2*.*

Example 24. The Quantified Boolean Formula (QBF) is the problem of the validity of a boolean formula with quantifiers over propositional variables. It is well-known to be PSPACE complete. Without loss of generality, we restrict formulae to $\neg, \vee, \exists$. It can be solved by the following rules:

$$\begin{array}{rlrlrl}
\mathtt{main}(\phi) & \to \mathtt{ver}(\phi, \mathbf{nil}) & \mathtt{not}(\mathbf{tt}) & \to \mathbf{ff} & \mathbf{0} = \mathbf{0} & \to \mathbf{tt} \\
\mathtt{in}(x, \mathbf{nil}) & \to \mathbf{ff} & \mathtt{not}(\mathbf{ff}) & \to \mathbf{tt} & \mathbf{S}(x) = \mathbf{0} & \to \mathbf{ff} \\
\mathtt{in}(x, \mathbf{cons}(a, l)) & \to \mathtt{or}(x = a, \mathtt{in}(x, l)) & \mathtt{or}(\mathbf{tt}, x) & \to \mathbf{tt} & \mathbf{0} = \mathbf{S}(y) & \to \mathbf{ff} \\
 & & \mathtt{or}(\mathbf{ff}, x) & \to x & \mathbf{S}(x) = \mathbf{S}(y) & \to x = y
\end{array}$$

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

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

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

They admit the following quasi-interpretations (of kind 0):

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

So the function is PSPACE-computable.

5 Further results

New results can be obtained by changing the semantics, the orderings or the quasi-interpretations. Firstly, let us consider non-confluent programs.

Theorem 25. *Functions computed by non confluent programs that admit a quasi-interpretation (of kind* 0*) and a MPO proof of termination are exactly* PSPACE *functions.*

This result is quite surprising. Indeed, by adding non-confluence (that is, non-determinism) to PTIME, we expected to characterise NPTIME. Here, we reach a step above and characterise PSPACE (actually, we do characterize NPSPACE, but these two classes are equal). Moreover, at this point, termination by MPO or LPO doesn't change the class characterised whereas it has always been the case otherwise.

Secondly, what can be said about the programs when we have no ideas on the kind of proof of termination. The following example shows that the question is not straightforward:

$$(1) \quad \frac{x\sigma = v}{\mathcal{E}, \sigma \vdash \langle C, x\rangle \to \langle C, v\rangle} (Variable)$$

$$(2) \quad \frac{\mathbf{c} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \to \langle C_n, \mathbf{c}(v_1, \cdots, v_n)\rangle} (Constructor)$$

$$(3) \quad \frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle \quad (\mathtt{f}(v_1, \cdots, v_n), v) \in C_n}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \to \langle C_n, v\rangle} (Cachereading)$$

$$(4) \quad \frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad p_i\sigma' = v_i \quad \mathcal{E}, \sigma' \vdash \langle C_n, r\rangle \to \langle C, v\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \to \langle C \bigcup (\mathtt{f}(v_1, \cdots, v_n), v), v\rangle} (Cachepus$$

Figure 1: Evaluation of a rewriting system with memorization of intermediate evaluations

Example 26.

$$\begin{array}{rcl} g(\epsilon) & \to & \epsilon \\ f(n) & \to & \mathtt{if}\ n = \epsilon \\ & & \quad \mathtt{then}\ \epsilon \\ & & \quad \mathtt{else}\ g(g(f(p(n)), f(p(n))), g(f(p(n)), f(p(n)))) \\ p(1(\epsilon)) & \to & \epsilon \\ p(0(i(x))) & \to & 1(p(i(x))) \\ p(1(i(x))) & \to & 0(i(x)) \end{array}$$

where $i \in \{0, 1\}$. Observe that $[\![p]\!]$ is the predecessor function on binary words (written from lower bits to higher bits). As a consequence, the system ends. It has a quasi-interpretation: $(\!|p|\!)(X) = (\!|f|\!)(X) = X$, $(\!|g|\!)(X, Y) = (\!|=|\!)(X, Y) = \max(X, Y)$, $(\!|1|\!)(X) = (\!|0|\!)(X) = X + 1$, $(\!|\epsilon|\!) = 1$. But, the derivation length of the system is of the order 2^{2^n}, and even by using a cache, one needs an exponential size. These are the bad news. However, one may note that the systems compute the function $w \mapsto \epsilon$! And this function is in Pspace. The general case keeps open. Nevertheless, we have the following restricted result.

Definition 27. A *strict quasi-interpretation* is a quasi-interpretation where for any symbols f, one has $(\!|f|\!)(X_1, \ldots, X_k) > X_i$ for all $i \leq n$.

Strict quasi-interpretation still don't ensure termination.

Theorem 28. *The set of functions computed by rewrite systems such that:*

- *it admits a strict quasi-interpretation,*
- *it terminates*

is exactly the set of Pspace *functions.*

References

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

[2] Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal Zilio, and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. Technical report, LIF 17-2004, January 2004.

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

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

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

[6] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Computer Science Logic, 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384, 1998.

[7] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. On lexicographic termination ordering with space bound certifications. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *PSI*, volume 2244 of *Lecture Notes in Computer Science*, pages 482–493. Springer, July 2001.

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

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

[10] Adam Cichon. Bounds on derivation lengths from termination proofs. Technical report, University of London, Royal Holloway and Bedford New College, 1990.

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

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Massachussetts Institut of Technology, 1990.

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

[14] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. Elsevier Science Publishers B. V. (NorthHolland), 1990.

[15] Jrgen Giesl. Generating polynomial orderings for termination proofs. In *Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 427–431, 1995.

[16] Erich Grädel and Yuri Gurevich. Tailoring Recursion for Complexity. *Journal of Symbolic Logic*, 60(3):952–969, September 1995.

[17] Dieter Hofbauer. Termination proofs with Multiset Path Orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

[18] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.

[19] S. Kamin and J.-J. Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerit de l'Illinois, Urbana, 1980. unpublished note.

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

[21] D. S. Lankford. On proving term rewriting systems are Noetherien. Technical Report MTP-3, Louisiana Technical University, 1979.

[22] Daniel Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[23] Daniel Leivant and Jean-Yves Marion. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae*, 19(1,2):167,184, September 1993.

[24] Jean-Yves Marion. Complexité implicite des calculs, de la thorie la pratique, 2000. Habilitation.

[25] Jean-Yves Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183, 2003.

[26] Jean-Yves Marion and Jean-Yves Moyen. Efficient First Order Functional Program Interpreter with Time Bound Certifications. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 25–42. Springer, November 2000.

[27] Jean-Yves Moyen. System Presentation: An analyser of rewriting systems complexity. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001. Workshop RULE, accessible `http://www.loria.fr/~moyen/`.

[28] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical report, Department of Computer Science, University of Illinois, 1978.

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

[30] Andreas Weiermann. Termination proofs by Lexicographic Path Orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.