Quasi-Interpretations and Small Space Bounds

Guillaume Bonfante[1], Jean-Yves Marion[1], and Jean-Yves Moyen[2]

[1] Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, and École Nationale Supérieure des Mines de Nancy, INPL, France.
[2] Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France, and Université Henri Poincaré Nancy I, France.

Abstract. Quasi-interpretations are an useful tool to control resources usage of term rewriting systems, either time or space. They not only combine well with path orderings and provide characterizations of usual complexity classes but also give hints in order to optimize the program. Moreover, the existence of a quasi-interpretation is decidable.
In this paper, we present some more characterizations of complexity classes using quasi-interpretations. We mainly focus on small space-bounded complexity classes. On one hand, by restricting quasi-interpretations to sums (that is allowing only affine quasi-interpretations), we obtain a characterization of LinSpace. On the other hand, a strong tiering discipline on programs together with quasi-interpretations yield a characterization of LogSpace.
Lastly, we give two new characterizations of Pspace: in the first, the quasi-interpretation has to be strictly decreasing on each rule and in the second, some linearity constraints are added to the system but no assumption concerning the termination proof is made.

1 Introduction

The quasi-interpretation over reals method is a decidable procedure to certify time or space complexity of a program based on rewriting semantics. This paper focuses on several small space complexity classes.

From a practical point of view, the bottom line is to perform a complexity analysis in order to measure space resources by static analysis. The knowledge of the resource consumption is an important information to compile more efficiently a program by certifying the space usage, this may also reinforce security. Dynamic approaches, like direct evaluation of the space or time usage by numerical recipes, as done by Benziger [1] for Nuprl, are the other way for controlling resources. The advantage of static analysis is that it can provide a certificate which guarantees the resource usage.

A program is a term rewriting system in which there are constructor terms defining the domain and the range of the computation. Quasi-interpretations are related to interpretation termination proofs. An interpretation is a quasi-interpretation. Conversely, a quasi-interpretation does not prove termination *a priori*. There are some interests to study programs, which have a polynomially bounded quasi-interpretation because it delineates a not too big class of functions. Surprisingly, a function, which admits a polynomial interpretation, can run in doubly exponential time. This observation was made by Hofbauer and Lautemann [2] and Cichon and Lescanne [3]. The whole picture about polynomial interpretation has been clarified in [4]. One of the key ideas is to separate constructor terms from defined function symbols. In fact, the program complexity depends mainly on the quasi-interpretation of constructor terms.

From the point of view of complexity, quasi-interpretations are very exciting. Indeed quasi-interpretations give an upper bound on the size of the normal form of a term if it exists. We obtained different resource upper bounds by combining quasi-interpretations and termination proofs.

– If the program admits a quasi-interpretation and terminates by Product Path Ordering, (PPO) then it is computable in polynomial time. See [5].
– If the program admits a quasi-interpretation and terminates by Lexicographic Path Ordering (LPO), then the computation consumes a polynomial space. See [6].

The results mentioned above should be compared to the expressive power of both recursive path ordering PPO and LPO. Hofbauer [7] showed that Multiset Path Ordering (MPO) gives rise to a characterization of primitive recursive functions. It is not difficult to see that the same proof goes for PPO instead of MPO. Then, Weiermann [8] has shown that LPO characterizes the multiple recursive functions. So, quasi-interpretations allow us to tame the complexity of treated algorithms.

In this paper, we present new characterizations of small space complexity classes. The first result is a characterization of the class LinSpace of functions, which are computable in linear space in the size of the inputs. For this, we restrict quasi-interpretations to purely affine functions, that is the quasi-interpretation of each symbol is a function like $aX + b$. The class of functions, which are computed by programs which terminate by LPO and admit a purely affine quasi-interpretations, is exactly LinSpace. Let us mention here the first intrinsic characterization of LinSpace. It is due to Ritchie [9].

The second result is a characterization of the class LogSpace which contains all functions which run in logarithmic space in the size of the inputs. Intuitively, this means that we cannot copy an input on some registers. However, we can use pointers to read and mark inputs. For this, we introduce the notion of tiers inside a rewrite system which has yet be suggested by Beckmann-Weiermann [10] and Marion [11]. The tiering discipline is due to independently Bellantoni-Cook [12] and to Leivant-Marion [13, 14] in the context of simply typed lambda-calculus. Our characterization of LogSpace is based on Jones [15] and is also related to the work of Gurevich [16] on primitive recursive function algebra over first order finite structures.

The last result is two characterizations of Pspace, which is the class of functions computable in polynomial space. Such characterizations of Pspace by functional programming have been already considered by Oitavem [17].

In this paper, we demonstrate the use of quasi-interpretations to bound memory during a program execution. These results do not give a meaningful upper bound on the runtime. However, it is interesting to have an upper bound on the memory in several practical cases related to embedded systems. In this direction, Amadio and al [18] applied those ideas successfully to resource bytecode verifier and to resource verifier for synchronous cooperative threads [19]. We consider quasi-interpretation over the reals as it gives us a procedure to synthesis quasi-interpretations. See [20, 21] for full details.

2 First Order Functional Programming

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

2.1 Syntax of Programs

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

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

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

Remark 2. Notice that function symbols (sometimes called defined symbols) do not appear in any pattern matching and may not have arity 0.

In the following, we will write $\overrightarrow{t}$ to denote the vector of terms $t_1, \cdots, t_n$ if those terms are not relevant for the current discussion.

Definition 3. *A* program *is a quadruple $\texttt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ 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 $\texttt{f}$.*

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

2.2 Semantics

The domain of the computed functions is the constructor algebra $\mathcal{T}(\mathcal{C})$. The set of rules induces a rewriting relation $\to$. We use $\xrightarrow{+}$ for the transitive closure of $\to$ and $\xrightarrow{*}$ for the reflexive transitive closure of $\to$. $t \xrightarrow{!} s$ iff $t \xrightarrow{*} s$ and s is in normal form.

Definition 4. *If the system is confluent, then we say that the program is* deterministic.

In this case, the program $\mathtt{f}$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ defined as follows. For all $u_i \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{f}]\!](\overrightarrow{u}) = v$ iff $\mathtt{f}(\overrightarrow{u}) \xrightarrow{!} v$ and v is a constructor term. Otherwise $[\![\mathtt{f}]\!](\overrightarrow{u})$ is undefined. See, among others, Huet [22] for conditions ensuring the confluence of the system.

Definition 5. *If the system is not confluent, then we say that the program is* non-deterministic.

In this case, following Grädel and Gurevich [23] or Bonfante, Cichon, Marion and Touzet [4], the value of any term is the maximum value of the normal forms of the terms (for a given order on terms, usually lexicographic ordering). Notice that this includes the usual definition for decision problems by choosing **true** $>$ **false**.

3 Orderings

3.1 Precedences and Ranks

Definition 6. *Let $f = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. A* precedence *over f is a pre-order $\preceq_{\mathcal{F}}$ over $\mathcal{F}$* compatible *with the rules. By compatible, we mean that if $g(\overrightarrow{p}) \to r$ is a rule, then for any symbol h in r, we have $h \preceq_{\mathcal{F}} g$. Let us note $\approx_{\mathcal{F}}$ the induced equivalence relation, that is $g \approx_{\mathcal{F}} h$ iff $g \preceq_{\mathcal{F}} h \wedge h \preceq_{\mathcal{F}} g$. The strict induced ordering is noted $\prec_{\mathcal{F}}$, $g \prec_{\mathcal{F}} h$ iff $g \preceq_{\mathcal{F}} h \wedge \neg(h \preceq_{\mathcal{F}} g)$.*

We extend canonically the ordering on $\mathcal{F} \bigcup \mathcal{C}$ by choosing $\mathbf{c} \prec_{\mathcal{F}} f$ for each constructor $\mathbf{c}$ and each function f and two constructors are mutually incomparable.

Lemma 7. *Let f be a program and $\preceq_{\mathcal{F}}$ be a precedence for that program. $\preceq_{\mathcal{F}}$ induces a* rank function *rk assigning an integer to each symbol such that $b \preceq_{\mathcal{F}} b'$ implies $rk(b) \leq rk(b')$ and $b \prec_{\mathcal{F}} b'$ implies $rk(b) < rk(b')$.*

Proof. By induction on the (finite) ordering. □

In the following, we will often speak indifferently of either precedence or rank function, depending whether we want to focus on the ordering or on the "slicing" aspect.

Proposition 8. *Let $f = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. There exists a precedence $\preceq_{\mathcal{F}}$ for that program.*

Proof. For each rule $\mathtt{g}(\overrightarrow{p}) \to r$ and each function symbol $\mathtt{h} \in r$, put $\mathtt{h} \preceq_{\mathcal{F}} \mathtt{g}$. Build the reflexive transitive closure of this relation. □

Thus, mutually recursive functions will have the same rank and subfunctions will have smaller rank than the caller.

Proposition 9. *Let f be a program and $\preceq_{\mathcal{F}}$ a precedence of f. Let $g(\overrightarrow{v}) \xrightarrow{*} t$ be such that the v_is are constructor terms. Then no symbol in t has a rank strictly higher than g, that is for all symbols $b \in t$, $b \preceq_{\mathcal{F}} g$.*

It is an immediate consequence of the compatibility and the transitivity of the precedence. This proposition is really important in the following because it allows some kind of computation "by rank": functions of high ranks will never interfere when computing the value of functions of low ranks. Thus, we will be able to design recursive procedure to compute functions depending on their ranks.

3.2 Termination Orderings

We now focus on termination orderings which we see as a kind of mold that capture some algorithmic patterns. Here, we consider Lexicographic Path Ordering which is a simplification ordering and so well-founded. See Dershowitz [24] for complete details about termination orderings. Krishnamoorthy and Narendran in [25] have proved that deciding whether a program terminates by Lexicographic Path Ordering is a NP-complete problem.

Let $\preceq$ be a partial ordering and $\prec$ be its strict part.

Definition 10. *The* lexicographic extension *of $\prec$, noted $\prec^l$, is defined as follows.*
We have $(m_1, \cdots, m_k) \prec^l (n_1, \cdots, n_{k'})$ if and only if there exists an index j such that (i) $\forall i < j, m_i \preceq n_i$ and (ii) $m_j \prec n_j$.

Lexicographic Path Ordering

Definition 11. *Given a precedence $\preceq_{\mathcal{F}}$, the* Lexicographic Path Ordering *(LPO) $\prec_{lpo}$ is defined in Figure 1.*

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

3.3 Quasi-Interpretations

In order to control resources more precisely that termination orderings do, we suggest the use of quasi-interpretations. Quasi-interpretations have been introduced by Bonfante, Marion [11] and Marion-Moyen [5] and have proved themselves useful to obtain complexity bounds on programs.

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

Definition 12 (Assignment). *An* assignment *of a symbol $b \in \mathcal{F} \bigcup \mathcal{C}$ whose arity is n is a function $(\!|b|\!) : (\mathbf{R}^+)^n \to \mathbf{R}^+$ such that:*

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

$$\frac{\forall i, s_i \preceq_{lpo} t_i \quad \exists j / s_j \prec_{lpo} t_j}{\mathbf{c}(s_1, \cdots, s_n) \prec_{lpo} \mathbf{c}(t_1, \cdots, t_n)} \mathbf{c} \in \mathcal{C} \qquad \frac{s = t_i \text{ or } s \prec_{lpo} t_i}{s \prec_{lpo} \mathtt{f}(\ldots, t_i, \ldots)} \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C}$$

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

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

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

Fig. 1. Definition of $\prec_{lpo}$

(Additive) *For each constructor* $\mathbf{c}$, $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}}$, $\alpha_{\mathbf{c}} \geq 1$

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

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

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

We have $(\!|s|\!) \geq (\!|t|\!)$ if t is a subterm of s. We have also, for every substitution σ, $(\!|s|\!) \geq (\!|t|\!)$ implies that $(\!|s\sigma|\!) \geq (\!|t\sigma|\!)$.

Definition 13 (Quasi-interpretation). *A* program assignment $(\!|-|\!)$ *is an assignment of each program symbol. An assignment* $(\!|-|\!)$ *of a program is a* quasi-interpretation *if for each rule* $l \to r$, $(\!|l|\!) \geq (\!|r|\!)$

It is worth noticing that the above inequality is not strict which differs from the notion of interpretation used to prove termination. See, among others, [4].

Note that we restrict ourselves to additive quasi-interpretations, that is the form of the assignment of constructors is fixed. Allowing other (polynomial) assignment to constructors brings higher upper bound on complexity as shown in [21].

Definition 14. *The quasi-interpretation of a function symbol* $\mathtt{f}$ *is said to be: (i)* affine *if* $(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} \alpha_{\mathtt{f},i} X_i + \alpha_{\mathtt{f}}$ *with* $\alpha_{\mathtt{f}} > 0$*, (ii)* multiplicative *if* $(\!|\mathtt{f}|\!)$ *is a polynomial of the form* $P + \alpha_{\mathtt{f}}$ *with* $\alpha_{\mathtt{f}} > 0$ *and* $P(x_1, \ldots, x_n) \geq x_i$ *for all* $i \leq n$.

The quasi-interpretation of a program is said to be purely affine *(resp.* purely multiplicative*) if the quasi-interpretation of each function symbol is affine (resp. multiplicative).*

Notice that for function symbols, even if $\alpha_{\mathtt{f}} < 1$, the quasi-interpretation may be affine. Whereas for constructors, we force $\alpha_{\mathtt{c}} \geq 1$.

Proposition 15 ([4]). *Suppose that* $\mathtt{f}$ *is a program with a quasi-interpretation. Assume* t *is a constructor term in* $\mathcal{T}(\mathcal{C})$*, we have* $|t| \leq (\!|t|\!) \leq O(|t|)$*.*

4 Call-Trees

We present now call-trees which are a tool that we shall use all along. Let $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. A call-tree gives a static view of an execution and captures all function calls. Hence, we can study dependencies between function calls without taking care of the extra details provided by the underlying rewriting relation. Call-trees are related to dependency graphs proposed by Aarts and Giesl [26].

Definition 16. *A* state *is a tuple* $\langle h, u_1, \cdots, u_p \rangle$*, sometimes noted* $\langle h, \overrightarrow{u} \rangle$*, where* h *is a function symbol of arity* p *and* $u_1, \ldots, u_p$ *are constructor terms. Assume that* $\eta_1 = \langle h, u_1, \cdots, u_p \rangle$ *and* $\eta_2 = \langle g, s_1, \cdots, s_m \rangle$ *are two states. A* transition *is a triplet* $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ *such that:*

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

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

Definition 17. *The* $\langle f, t_1, \cdots, t_n \rangle$*-call tree is a tree defined as follows: (i) nodes are states, (ii) the root is the state* $\langle f, t_1, \cdots, t_n \rangle$*, (iii) for each state* η_1*, the children of* η_1 *are the states* $\{\eta_2 \mid \eta_1 \overset{e}{\rightsquigarrow} \eta_2 \in Transition(f)\}$*.*

Proposition 18. *Assume that* $\eta_1 = \langle h, \overrightarrow{u} \rangle \overset{*}{\rightsquigarrow} \eta_2 = \langle g, s_1, \cdots, s_m \rangle$ *and* $(\!|-|\!)$ *is a quasi-interpretation for that program. Then we have* $(\!|g(s_1, \cdots, s_m)|\!) \leq (\!|h(\overrightarrow{u})|\!)$ *and thus* $|s_i| \leq (\!|s_i|\!) \leq (\!|h(\overrightarrow{u})|\!)$ *for all* $1 \leq i \leq m$*.*

Proof. By virtue of the quasi-interpretation definition. Both the subterm property, the weak monotonicity property and Proposition 15 are necessary. □

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

Lemma 19. *The size of each state of the* $\langle f, t_1, \cdots, t_n \rangle$*-call tree is bounded by* $d \times (\!|f(\overrightarrow{t})|\!)$ *where* d *is the maximal arity of a function symbol.*

Proof. $|\langle \mathtt{g}, s_1, \cdots, s_m \rangle| = \sum_i |s_i| \leq \sum_i (\!|s_i|\!) \leq m \times (\!|\mathtt{f}(\overrightarrow{t})|\!) \leq d \times (\!|\mathtt{f}(\overrightarrow{t})|\!)$ □

5 Linear Space

Let a $\mathrm{LPO}^{\mathrm{QI}+}$-program be a program that terminates by LPO and admits a purely affine quasi-interpretation.

Theorem 20. *The set of functions computed by deterministic LPO^{QI+}-programs is exactly* LinSpace.

Proof. The upper-bound on complexity is established by Theorem 27. The completeness of the characterization is established by Theorem 25. $\square$

Theorem 21. *The set of functions computed by non-deterministic LPO^{QI+}-programs is exactly* NLinSpace.

5.1 Unary k-Stack Turing Machines

In this Section, we follow Gurevich [16], Leivant [13] and Marion et al. [4].

Lemma 22 ([4]). *A function is in* LinSpace *if it is computed by a multi-stack Turing Machine (STM) over the unary alphabet in polynomial time.*

Actually, we will use another characterization.

Lemma 23. *A function is in* LinSpace *if it is computed by a polynomial time multi-stack Turing Machine over the unary alphabet and each stack is bounded by the size of the input.*

Proof. Following Lemma 22, we take M a STM that computes a function $f \in$ LinSpace. Let K be such that M works in time less than n^K. As a consequence, each stack is bounded by n^K. We can encode such numbers by K stacks, each representing a digit of the number in base n. Suppose that M has p stacks, let N have $2Kp$ stacks $\Pi_1^1, {\Pi'}_1^1, \ldots, \Pi_K^1, {\Pi'}_K^1, \ldots, \Pi_1^p, {\Pi'}_1^p, \ldots, \Pi_K^p, {\Pi'}_K^p$. The stacks $\Pi_1^1, \Pi_2^1, \ldots, \Pi_K^1$ of N represent the stack Π_1 of M. The primed stacks are the complement of their corresponding stack to n, the length of the input. To simulate the incrementing of a stack of M, say Π_1, find the first non-empty stack ${\Pi'}_j^1$, decrement it by one, increment Π_j^1 by one, then reset all Π_i^1 to 0 and all ${\Pi'}_i^1$ to n where $i < j$. Decrementing a stack is done in a similar way. So, the simulation is done within the space bound. $\square$

5.2 Simulation of Linspace by $\mathbf{LPO}^{\mathbf{QI}+}$-programs

Given a function $f : \mathbb{W}^n \to \mathbb{W}$ in LinSpace, we take a p-stack Turing Machine M which computes f, works on the unary alphabet $\{s\}$ and such that each stack is bounded by the length of the input. Say that M works in time n^k for some k. As we do the simulation of the machine step for step, we need to design a program whose derivation length is at least n^k on input n.

Lemma 24. *Given $k \in \mathbf{N}$, there is a (deterministic) LPO$^{QI_+}$-programs that runs in n^k steps on input of size n.*

Proof. Let us first show how to have a program whose derivation length is the square of its argument.

$$\begin{aligned}
\mathtt{C}_2(\mathbf{0},\mathbf{0},y') &\to \mathbf{0}\\
\mathtt{C}_2(\mathbf{S}(x),\mathbf{0},y') &\to \mathtt{C}_2(x,y',\mathbf{0})\\
\mathtt{C}_2(x,\mathbf{S}(y),y') &\to \mathtt{C}_2(x,y,\mathbf{S}(y'))\\
\mathtt{P}_2(x) &\to \mathtt{C}_2(x,x,\mathbf{0})
\end{aligned}$$

Let us define $(\!|\mathbf{0}|\!) = 1$ and $(\!|\mathbf{S}|\!)(X) = X + 1$. By choosing $(\!|\mathtt{C}_2|\!)(X,Y,Y') = X + Y + Y' + 1$ and $(\!|\mathtt{P}_2|\!)(X) = 2X + 2$, the program admits a purely affine quasi-interpretation. It is routine to check that it terminates by LPO.

Now, to have a program whose derivation length is n^k, we use the same trick: have a program $\mathtt{C}_k$ with $2k$ (actually $2k-1$ is sufficient) variables x_i, x'_i and adapts the rules of $\mathtt{C}_2$ to allow $\mathtt{C}_k$ to count in base n. x'_i is always the complementary of x_i to n, thus allowing to reinitialize the values. The generic rule is:

$$\mathtt{C}_k(\mathbf{S}(x_1),x'_1,\ldots,\mathbf{S}(x_i),x'_i,\mathbf{0},x'_{i+1},\ldots) \to \mathtt{C}_k(\mathbf{S}(x_1),x'_1,\ldots,x_i,\mathbf{S}(x'_i),x'_{i+1},\mathbf{0},\ldots)$$

Then, $\mathtt{P}_k(x)$ simply calls $\mathtt{C}_k$ with each $x_i = x$ and $x'_i = \mathbf{0}$. The resulting program has a derivation length equal to n^k, $(\!|\mathtt{C}_k|\!)(X_1,\ldots,X_{2k}) = \sum_{i=1}^{2k} X_i + 1$ and $(\!|\mathtt{P}_k|\!)(X) = kX + k + 1$, both are purely affine for a fixed k.

□

Theorem 25. *Suppose that f is in* LinSpace*, then it is computable by a LPO$^{QI_+}$-program.*

Proof. Let M be a p-Stack Turing machine computing f. As Lemma 23 shows it, we can suppose that the stacks are bounded by the size of the input along the computation. Let us consider the following system. To the template rules above, we add some new variables, one variable for the state of the machine and two for each stack of M. The first one (Π_i) represents the content of the stack (the ith), the second one (Π'_i) its complement to n, the size of the input.

Take now for instance a transition that increments the stack Π_1 and goes to state q' if Π_2 is empty in state q, we transform each rule $\mathtt{C}_k(\overrightarrow{x}) \to \mathtt{C}_k(\overrightarrow{y})$ as follows:

$$\mathtt{C}_k(\overrightarrow{x},\mathbf{q},\Pi_1,\mathbf{S}(\Pi'_1),\mathbf{0},\Pi'_2,\ldots) \to \mathtt{C}_k(\overrightarrow{y},\mathbf{q}',\mathbf{S}(\Pi_1),\Pi'_1,\mathbf{0},\Pi'_2,\ldots)$$

The program terminates by LPO because there is a decrease between $(\overrightarrow{x})$ and $(\overrightarrow{y})$. $(\!|\mathtt{C}_k|\!)$ is the sum of its arguments. □

5.3 Computation of LPO$^{\mathbf{QI}+}$-Programs in linear space

Let $\mathtt{f}$ be a program admitting a purely affine quasi-interpretation. If each function symbol $\mathtt{f}$ has a quasi-interpretation $(\!|\mathtt{f}|\!)(X_1, \cdots, X_n) = \sum \alpha_{\mathtt{f},i} X_i + \alpha_{\mathtt{f}}$ with $\alpha_{\mathtt{f}} > 0$, then the following proposition holds:

Proposition 26. *The size of any term t is bounded by $K(\!|t|\!)$ for some constant K.*

Proof. Take $\alpha = \min_{a \in \mathcal{F} \cup \mathcal{C}}(\alpha_a)$. As $\mathcal{F} \cup \mathcal{C}$ is finite, $\alpha > 0$. By induction, we have: $|t| \leq (\!|t|\!)/\alpha$.

Theorem 27. *LPO^{QI+}-Programs are computable by* LinSpace *machines.*

Proof. First of all notice that if $\mathtt{f}(\overrightarrow{t}) \xrightarrow{*} s$, then $(\!|s|\!) \leq (\!|\mathtt{f}(\overrightarrow{t})|\!)$. At the same time, we have $|s| \leq K(\!|s|\!)$ with respect to Proposition 26. And finally, Proposition 15 shows that $(\!|\mathtt{f}(\overrightarrow{t})|\!) \leq (\!|\mathtt{f}|\!)(B(|\overrightarrow{t}|) + 1)$ for some constant B. But, as $(\!|\mathtt{f}|\!)$ is affine, we have $(\!|\mathtt{f}(\overrightarrow{t})|\!) \leq (\!|\mathtt{f}|\!)(B(|\overrightarrow{t}|) + 1) \leq A(|\overrightarrow{t}| + 1)$ for some constant A.

Putting things together, we conclude that the size of the terms obtained along the computation is bounded by: $|s| \leq KA(|t| + 1)$.

A reduction step may be performed within linear space and this ensure that the whole computation can. Indeed, finding a redex needs finite memory (plus a pointer in the term) and replacing the redex may take finitely many times the size of the input if one has to recopy some variables, like for instance in the rule: $\mathtt{D}(x) \to \mathtt{add}(x, x)$. So, the linear constant of the cost of replacing a term can be bounded by: $M = \max_{l \to r \in \mathcal{E}}(|r|)$. In other words, the computation can be done in space $AMKn$ on inputs of size n. ◻

6 Logarithmic Space

From now on, we do not make any supposition concerning the quasi-interpretation of functions. This means that the quasi-interpretation of a function must only respect the rules (Subterm) and (Weak Monotonicity) of Definition 12 and this allow, for example, $(\!|\mathtt{f}|\!)(X, Y) = \max(X, Y)$.

In this section, we follow the work of Jones [15] in order to get a characterization of LogSpace. The arguments of a function are separated into three distinct sets:

- An input set. This set is read-only and contains binary words that are subterms of the initial inputs.
- A working set. This set is read-write and contains unary integers. This condition, together with the quasi-interpretation, has the net effect of allowing these variables to be only pointers on a part of the input.
- An output set. This set is write-only and ensures that the result of a recurrence is not allowed to control another recurrence (at the same rank). This is done by preventing any pattern matching on this set. These conditions are forced by a tiering-rule similar to the one used by Bellantoni and Cook [12].

We consider programs whose inputs are binary words (that is built from $\{\boldsymbol{\epsilon}, \mathbf{0}, \mathbf{1}\}$) and outputs are unary integers (that is built from $\{\mathbf{Z}, \mathbf{S}\}$). We now define a strong tiering discipline on those programs, separating arguments of functions into three sets: inputs, working and simple as described above.

Definition 28. *Formally, let $LinPO^{QI}_{tier}$ be the set of programs f such that:*

- *(Cons)* $\mathcal{C} = \{\boldsymbol{\epsilon}, \boldsymbol{0}, \boldsymbol{1}, \boldsymbol{Z}, \boldsymbol{S}\}$;
- *(LPO)* *f terminates by LPO.*
- *(QI)* *f admits a quasi-interpretation.*
- *(Lin)* *If $\mathsf{f}(\overrightarrow{u}) \to t$, there is at most one function symbol with the same precedence as f in t. That is $\#\{\mathsf{g} \in \mathcal{F}, \mathsf{g} \approx_{\mathcal{F}} \mathsf{f} \text{ and } \mathsf{g} \in t\} \leq 1$.*

Second, it is tiered, that is

- *(Rule)* *Each rule has the form $\mathsf{f}(p_1, \cdots, p_n; q_1, \cdots, q_m; y_1, \cdots, y_l) \to t$ where p_i is a pattern over the binary words, q_i is a pattern over the unary integers and y_i is a variable.*
- *(Input)* *If there is a subterm $\mathsf{g}(p'_1, \cdots, p'_k; \overrightarrow{q'}; \overrightarrow{y'}) \trianglelefteq t$ then for all i, there exists j such that $p'_i \trianglelefteq p_j$.*
- *(Output)* *If there is a subterm $\mathsf{g}(\overrightarrow{p'}; q'_1, \cdots, q'_{m'}; y'_1, \cdots, y'_{l'}) \trianglelefteq t$, then neither $\boldsymbol{0}, \boldsymbol{1}, \boldsymbol{\epsilon}$ nor a variable present in any p'_i appears in any of the q'_i or the y'_j.*
- *(Tier)* *If there is a subterm $\mathsf{g}(\overrightarrow{p'}; q'_1, \cdots, q'_{m'}; \overrightarrow{y'}) \trianglelefteq t$ then for all i and j, $y_j \notin q'_i$.*
- *(Simple)* *If there are two subterms $\mathsf{g}(\overrightarrow{p'}; q'_1, \cdots, q'_{m'}; \overrightarrow{y'}) \trianglelefteq t$ and $\mathsf{h}(\overrightarrow{p''}; \overrightarrow{q''}; \overrightarrow{y''}) \trianglelefteq q'_i$ then $\mathsf{h} \prec_{\mathcal{F}} \mathsf{g}$. That is the result of a function can only be matched by functions of strictly higher rank.*

The condition (Rule) separates the three sets of arguments for each function. The condition (Input) states that the input set is read-only. The condition (Output) says that inputs have not the same type as other variables and thus may no be used anywhere. The conditions (Tier) and (Simple) ensure the tiering discipline with respect to the precedence.

Theorem 29. *The set of functions computed by $LinPO^{QI}_{tier}$-programs is exactly* LogSpace*, the set of functions computable in logarithmic space on a Turing machine.*

We only give here the sketch of proof. A more complete proof can be found in [27].

Proof (Sketch of proof). Jones has shown that LogSpace is the set of functions computed by polynomial time counter machines with counter no bigger than the input [15]. It is easy to simulate such a machine by a $\mathrm{LinPO}^{\mathrm{QI}}_{\mathrm{tier}}$-program. The polynomial clock is done in a way similar to Lemma 24 and the counters can easily be stored into the read-write variables.

In order to show that $\mathrm{LinPO}^{\mathrm{QI}}_{\mathrm{tier}}$-programs can be computed in logarithmic space, there are crucial points:

– the constructor terms in input position are always subterms of the initial term. As a consequence, they can be stored by a pointer. This is due to Hypothesis (Input) ;
– the constructor terms in working and simple position have a polynomial size (wrt the input data) ; it is a consequence of Hypothesis (QI). As they are unary (Hypothesis (Output)), they also can be stored in logarithmic space.
– consider the deriving strategy where you rewrite the symbol of minimal rank that is leftmost innermost. The tiering discipline shows that this strategy is complete (Hypotheses (Tier, Simple)); that is give the normal form of the term. Notice that the system terminates due to Hypothesis (LPO).
– due to the Hypothesis (Lin), the rewriting strategy above is such that there are only finitely many function symbols in terms along the computation.

In other words, each terms along the computation can be coded in logarithmic space and as each step can be performed in logarithmic space, the computation can be done within the space bound.

□

7 Polynomial Space

7.1 Strict Quasi-Interpretations

We say that a program admits a strict-quasi-interpretation if it admits a quasi-interpretation and (i) each rule verifies: $(\!|l|\!) > (\!|r|\!)$, (ii) $(\!|\mathtt{f}|\!)(x_1, \ldots, x_n) \geq x_i + \alpha_{\mathtt{f}}$ for some constant $\alpha_{\mathtt{f}} > 0$ and all $i \leq n$.

Remark 30. If the interpretations are over rational numbers, condition (ii) can be reformulated: $(\!|\mathtt{f}|\!)(x_1, \ldots, x_n) > x_i$ for all i.

Recall that a polynomial interpretation verifies all the properties above. They have the further property: the interpretations are necessarily polynomials. This prevent in particular the use of max in assignments. In that case, we have:

Theorem 31 (Bonfante, Cichon, Marion, Touzet [4]). *Programs which admit polynomial interpretations compute exactly* PTIME.

Theorem 32 (Dershowitz [28]). *A program with a strict-quasi-interpretation terminates.*

Proposition 33. *The height of a term t of a program that admits a strict-quasi-interpretation is bounded by $(\!|t|\!)$.*

Theorem 34. *A function is computed by a program that admits a strict quasi-interpretation iff it is in* PSPACE.

Proof. The fact that a function in PSPACE can be computed by a strict-quasi-interpretation is shown by Lemma 60 of [21]. The quasi-interpretation given there need to be slightly modified in order to be strict.

Let $\mathtt{f}$ be a program which admits a strict quasi-interpretation and $\mathtt{f}(\overrightarrow{t})\xrightarrow{*}s$. The hypothesis on the assignment induces a polynomial bound on the height of terms. So, by considering a leftmost innermost strategy, we have a polynomial bound on the size of the term. Actually, there are at most $\alpha \times P(n)$ (with $P(n)$ the bound on the height of the term and α the maximal size of a right hand side of a rule) defined symbols in terms along the computation. As a consequence, the size of the term is $O(P(n)^2)$.

Computing the next configuration can be done clearly in polynomial space. As a consequence, the computation can be done within PSPACE.

□

There is actually an other characterization of PSPACE due to Bonfante [29]. It shows the role of the two parameters that we are considering here: the choice of $l > r$ or $l \geq r$ and the choice of interpretations over $+, \times$ or $+, \times, \max$.

Theorem 35 (Bonfante [29]). *Functions computed by programs such that*

1. *the program admits a purely multiplicative quasi-interpretation,*
2. *the program terminates,*

are exactly PSPACE *functions.*

7.2 Linear Systems

Definition 36. *Let* $\mathtt{f}$ *be a program and* $(\prec_{\mathcal{F}}, \approx_{\mathcal{F}})$ *be a precedence.* $\mathtt{f}$ *is* linear *with respect to* $(\prec_{\mathcal{F}}, \approx_{\mathcal{F}})$ *if for each rule* $\mathtt{f}(\overrightarrow{p}) \to r$ *there is at most one symbol in* r *with the same precedence as* $\mathtt{f}$.

When the precedence is unspecified, we implicitly refer to a precedence compatible with the program such as the one used for termination orderings or the one build in Proposition 9. In those cases, we simply say that the program is linear.

Let a LinQI-program be a program that (i) terminates, (ii) is linear with respect to the implicit precedence, and (iii) admits a quasi interpretation.

Theorem 37. *The set of functions computed by LinQI-programs is exactly* PSPACE.

Proof. The proof will goes by induction on the rank of the head symbol of nodes in a call-tree.

If the rank is 0 (which can be achieved by a trivial extension of the definition of call-trees), then no computations are needed in order to have the value of $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$ (remember that in a node, t_is are constructors terms).

Otherwise, we introduce sequents of the form $\langle \mathtt{f}, t_1, \cdots, t_n\rangle; t \vdash \langle \mathtt{g}, s_1, \cdots, s_m\rangle; s$ meaning "$\mathtt{f}(\overrightarrow{t})\xrightarrow{!}t$ if $\mathtt{g}(\overrightarrow{s})\xrightarrow{!}s$", and we define the covering graph of $\mathtt{f}$ in the $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$-call tree as the sub-graph obtained by keeping only nodes labelled by symbols with the same rank as $\mathtt{f}$.

As the program is linear, the covering graph is a chain. As the program admits a quasi-interpretation, it has at most $O(2^B)$ nodes, where $B = (\!|\mathtt{f}(\overrightarrow{t})|\!)$. Moreover, the size of each node of the call-tree is bounded by B.

To compute the value of $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$, proceed as follows:

1. Guess a node $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$ with $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ and $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$ having no child with the same precedence, guess two values t and s and introduce the sequent $\langle \mathtt{g}, s_1, \cdots, s_m\rangle; s \vdash \langle \mathtt{f}, t_1, \cdots, t_n\rangle; t$.
2. Perform a reduction step on $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$. If $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$ is a child of $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$ in the call-tree, then one just need to compute the values of other children (using s when necessary) to verify that t was correctly guessed. This can be done in polynomial space by induction hypothesis.
3. If $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$ is not a child of $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$, then guess a new node $\langle \mathtt{h}, u_1, \cdots, u_p\rangle$ and a new value u. Replace the current sequent by the two sequents $\langle \mathtt{h}, u_1, \cdots, u_p\rangle; u \vdash \langle \mathtt{f}, t_1, \cdots, t_n\rangle; t$ and
$\langle \mathtt{g}, s_1, \cdots, s_m\rangle; s \vdash \langle \mathtt{h}, u_1, \cdots, u_p\rangle; u$.

The depth of the proof tree for any sequent will be logarithmic in the length of the covering graph, thus polynomial in the size of the inputs. Each sequent has a polynomial size. Thus a clever depth-first search will allow to compute the value of a term in polynomial space. □

The three characterizations of PSPACE may be obtained as a corollary of the following conjecture.

Claim. The set of functions computed by programs which terminate and admit a quasi-interpretation is exactly PSPACE.

References

1. Benzinger, R.: Automated complexity analysis of NUPRL extracts. PhD thesis, Cornell University (1999)
2. Hofbauer, D., Lautemann, C.: Termination proofs and the length of derivations. In: RTA. Number 355 in Lecture Notes in Computer Science (1988)
3. Cichon, E., Lescanne, P.: Polynomial interpretations and the complexity of algorithms. In: CADE'11. Number 607 in Lecture Notes in Artificial Intelligence (1992) 139–147
4. Bonfante, G., Cichon, A., Marion, J.Y., Touzet, H.: Algorithms with polynomial interpretation termination proof. Journal of Functional Programming **11** (2000)
5. Marion, J.Y., Moyen, J.Y.: Efficient first order functional program interpreter with time bound certifications. In: LPAR 2000. Volume 1955 of Lecture Notes in Computer Science., Springer (2000) 25–42
6. Bonfante, G., Marion, J.Y., Moyen, J.Y.: On lexicographic termination ordering with space bound certifications. In: PSI 2001, Ershov Memorial Conference. Volume 2244 of Lecture Notes in Computer Science., Springer (2001)
7. Hofbauer, D.: Termination proofs with multiset path orderings imply primitive recursive derivation lengths. Theoretical Computer Science **105** (1992) 129–140
8. Weiermann, A.: Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. Theoretical Computer Science **139** (1995) 335–362
9. Ritchie, R.: Classes of predictably computable functions. Transaction of the American Mathematical Society **106** (1963) 139–173
10. Beckmann, A., Weiermann, A.: A term rewriting characterization of the polytime functions and related complexity classes. Archive for Mathematical Logic **36** (1996) 11–30

11. Marion, J.Y.: Analysing the implicit complexity of programs. Information and Computation **183** (2003) 2–18
12. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2** (1992) 97–110
13. Leivant, D.: Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Clote, P., Remmel, J., eds.: Feasible Mathematics II. Birkhäuser (1994) 320–343
14. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Fundamenta Informaticae **19** (1993) 167,184
15. Jones, N.D.: LOGSPACE and PTIME characterized by programming languages. Theoretical Computer Science **228** (1999) 151–174
16. Gurevich, Y.: Algebras of feasible functions. In: Twenty Fourth Symposium on Foundations of Computer Science, IEEE Computer Society Press (1983) 210–214
17. Oitavem, I.: A term rewriting characterization of the functions computable in polynomial space. Archive for Mathematical Logic **41** (2002) 35–47
18. Amadio, R., Coupet-Grimal, S., Zilio, S.D., Jakubiec, L.: A functional scenario for bytecode verification of resource bounds. In: CSL. (2004) to appear.
19. Amadio, R.M., Dal-Zilio, S.: Resource control for synchronous cooperative threads. In: CONCUR. (2004) 68–82
20. Bonfante, G., Marion, J.Y., Moyen, J.Y., Péchoux, R.: Synthesis of quasi-interpretations. Technical report, Loria (2005) Submited to RULE '05. Available at `http://www.loria.fr/~bonfante/publis/RULE05.pdf`.
21. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations. Technical report, Loria (2004) Submited to Theoretical Computer Science, accessible: `http://www.loria.fr/~moyen/`.
22. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems. Journal of the ACM **27** (1980) 797–821
23. Grädel, E., Gurevich, Y.: Tailoring recursion for complexity. Journal of Symbolic Logic **60** (1995) 952–969
24. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science **17** (1982) 279–301
25. Krishnamoorthy, M.S., Narendran, P.: On recursive path ordering. Theoretical Computer Science **40** (1985) 323–328
26. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science **236** (2000) 133–178
27. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations and small space bounds. Technical report, Loria (2005) `http://www.loria.fr/~bonfante/publis/RTA05f.pdf`.
28. Dershowitz, N.: A note on simplification ordering. Information Processing Letters **9** (1979) 212–215
29. Bonfante, G.: Constructions d'ordres, analyse de la complexité. Thèse, Institut National Polytechnique de Lorraine (2000)