# System Presentation: An Analyser of rewriting systems complexity

J.-Y. Moyen *

May 10, 2001

**Abstract**

This paper briefly describes ICAR, a program which analyses the implicit complexity of first order functionnal programs. ICAR is based on two previous characterisation of PTIME and PSPACE by mean of term rewriting termination orderings and polynomial quasi-interpretations.

## 1 The analyser ICAR

We shall consider term rewriting systems build over three distinct sets: function symbols (defined symbols), constructors and variables. The function symbols are ordered by a precedence $\prec_{\mathcal{F}}$ and constructors are considered as the smaller elements of the precedence. A program is a set of rewriting rules.

ICAR (Implicit Complexity AnalyseR) first checks the termination of the given rewriting system and then tries to find a bound on its complexity. This work is based on [3, 4, 1] for complexity analysis. Program transformation by the mean of memoization is based on Jones work [2].

- **Termination** is check using termination ordering, either the Multiset Path Ordering or the Lexicographic Path Ordering.
- **Complexity** may then be determined by combining the termination ordering used and quasi-interpretations. ICAR may be able to tell that the computed function is in PTIME or PSPACE.
- One of the main interest of our approach is that this analysis gives an upper bound on the complexity of the function computed rather than on the complexity of the program. This kind of complexity analysis was dubbed *implicit*. So ICAR also gives a way (*i.e.* a new operationnal semantics) to effectively achieve this bound.

One may then run the program using different operationnal semantics and verify experimentally the theoretical bound previously obtained.

*Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France, `moyen@loria.fr`

$$\frac{s \preceq_{mpo} t_i}{s \prec_{mpo} f(\ldots, t_i, \ldots)} \; f \in \mathcal{F} \bigcup \mathcal{C}$$

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

$$\frac{\{s_1, \cdots, s_n\} \prec^m_{mpo} \{t_1, \cdots, t_n\} \quad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g}}{\mathtt{g}(s_1, \cdots, s_n) \prec_{mpo} \mathtt{f}(t_1, \cdots, t_n)} \; \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

Figure 1: Multiset Path Ordering

# 2 Termination ordering and Quasi-interpretations

## 2.1 Termination orderings

**Definition 1.** Let $\prec$ be an ordering over terms. The *multiset extension* $\prec^m$ of $\prec$ is defined as follow:
$M = \{m_1, \cdots, m_k\} \prec^m \{n_1, \cdots, n_k\} = N$ if and only if $M \neq N$ and there exists a permutation$\pi$ such that:

- There exists $j$ such that $m_j \prec n_{\pi(j)}$
- For all $i$, $m_i \preceq n_{\pi(i)}$

**Definition 2.** The *Multiset Path Ordering* (MPO) is defined in the rules of Figure 1.

A MPO-program is a program such that for each rule $l \to r$, $r \prec_{mpo} l$.

**Definition 3.** Let $\prec$ be an ordering over terms. The *lexicographic extension* $\prec^l$ of $\prec$ is recursively defined as follow:

$(t_1, \cdots, t_n) \prec^l (s_1, \cdots, s_m)$ if and only if $t_1 \prec s_1$ or $t_1 = s_1$ and $(t_2, \ldots, t_n) \prec^l (s_2, \ldots, s_m)$.

**Definition 4.** The *Lexicographic Path Ordering* (LPO) is defined in the rules of Figure 2.

A LPO-program is a program such that for each rule $l \to r$, $r \prec_{lpo} l$.

$$\frac{s \preceq_{lpo} t_i}{s \prec_{lpo} \mathtt{f}(\dots, t_i, \dots)}$$

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

$$\frac{(s_1, \cdots, s_n) \prec^{l}_{lpo} (t_1, \cdots, t_n) \quad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \quad s_j \prec_{lpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{lpo} \mathtt{f}(t_1, \cdots, t_n)}$$

Figure 2: Lexicographic Path Ordering

## 2.2 Quasi-interpretation

**Definition 5.** A *quasi-interpretation* of a symbol $f$ is a function $(\!|f|\!)$ such that:

- $(\!|f|\!)$ is bounded by a polynomial
- $(\!|f|\!)$ is (non strictly) increasing
- $(\!|f|\!)(X_1, \cdots, X_n) \geq X_i$ for all $i \leq n$.
- $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \gamma$ for all constructors $\mathbf{c}$ where $\gamma > 0$ is a constant.

Quasi-interpretations are extended to terms as usual:

$$(\!|f(t_1, \cdots, t_n)|\!) = (\!|f|\!)((\!|t_1|\!), \dots, (\!|t_n|\!))$$

A program admits a quasi-interpretation if for each rule $l \to r$, $(\!|r|\!) \leq (\!|l|\!)$. This is clearly not sufficient for termination.

## 2.3 Theorems

**Theorem 6.** *The set of functions computable by MPO-programs admiting quasi-interpretations is* exactly PTIME, *the set of functions computable in polynomial time.*

*Proof.* See [4]. □

**Theorem 7.** *The set of functions computable by LPO-programs admiting quasi-interpretations is* exactly PSPACE, *the set of functions computable in polynomial space.*

*Proof.* See [1] □

## 3 Implementation

There are two things to implement. First, determining whether the program terminates by MPO or LPO and second determining if it admits a quasi-interpretation.

The termination using the orderings is a well-known problem. It is known to be PTIME computable if the precedence is given and NP-complete if the precedence has to be found. Fortunatly ICAR uses a restriction of the usual orderings: a constructor always has a precedence smaller than any function symbol.

**Claim 8.** *With this restriction, the precedence can be found in polynomial time. So termination by either MPO or LPO can be checked in time $2^{c\times k}$ where $k$ is the maximal arity of a function symbol and $c$ is a constant.*

*Proof.* See section 4 □

The main difficulty lies in the second part. Indeed, one doesn't know if quasi-interpretations are decidable. The similar problem with strict inegalities (*i.e.* finding polynomial interpretations) seems to be undecidable. Fortunatly, quasi-interpretations are quite easy to find because an upper bound on the program denotation turns to be a good candidate. So, even if finding one is a hard task for a computer, its an easy job for the programmer. The idea is to provide a potential quasi-interpretation together with the rewriting rules, and the program just has to check it.

Of course, there exists programs able to deal with symbolic computation. So the obvious way to check quasi-interpretations is to deleguate this job to such a program. Currently, ICAR uses Maple as a quasi-interpretation checker. Maple may be unable to check some inequality (especially those using a lot of *max*s. As far as I know, there isn't any software able to treat them properly, but one is under devellopement by Fabrice Rouiller at LIP6 (Paris) and should be available by the end of the year.

Until this moment, ICAR returns the quasi-interpretations that Maple was unable to solve and hopes that the user will be less clumsy.

**Example 9.**

1. the following program computes the addition and the multiplication of two unary numbers.

$$\begin{aligned}
\mathtt{add}(\mathbf{0},y) &\to y\\
\mathtt{add}(\mathbf{S}(x),y) &\to \mathbf{S}(\mathtt{add}(x,y))\\
\mathtt{mult}(\mathbf{0},y) &\to \mathbf{0}\\
\mathtt{mult}(\mathbf{S}(x),y) &\to \mathtt{add}(y,\mathtt{mult}(x,y))
\end{aligned}$$

It terminates either by MPO or LPO by putting $\mathtt{add} \prec_{\mathcal{F}} \mathtt{mult}$ and admits a quasi-interpretation: $(\!|\mathtt{add}|\!)(X,Y) = X+Y$, $(\!|\mathtt{mult}|\!)(X,Y) = X \times Y$, $(\!|\mathbf{0}|\!) = 1$, $(\!|\mathbf{S}|\!)(X) = X+1$. Let's verify the quasi-interpretaion for the last rule:

$$(\!|\mathtt{mult}(\mathbf{S}(x),y)|\!) = (\!|\mathtt{mult}|\!)((\!|\mathbf{S}(x)|\!),Y) = (X+1)\times Y$$
$$(\!|\mathtt{add}(y,\mathtt{mult}(x,y))|\!) = Y + (\!|\mathtt{mult}(x,y)|\!) = Y + X \times Y$$

2. one may computes the length of the longest common subsequence of two string as follow:

$$\begin{aligned}
\mathtt{max}(n,\mathbf{0}) &\to n\\
\mathtt{max}(\mathbf{0},m) &\to m\\
\mathtt{max}(\mathbf{S}(n),\mathbf{S}(m)) &\to \mathbf{S}(\mathtt{max}(n,m))
\end{aligned}$$

$$\begin{aligned}
\mathtt{lcs}(x,\boldsymbol{\epsilon}) &\to \mathbf{0}\\
\mathtt{lcs}(\boldsymbol{\epsilon},y) &\to \mathbf{0}\\
\mathtt{lcs}(\mathbf{a}(x),\mathbf{a}(y)) &\to \mathbf{S}(\mathtt{lcs}(x,y))\\
\mathtt{lcs}(\mathbf{b}(x),\mathbf{b}(y)) &\to \mathbf{S}(\mathtt{lcs}(x,y))\\
\mathtt{lcs}(\mathbf{a}(x),\mathbf{b}(y)) &\to \mathtt{max}(\mathtt{lcs}(x,\mathbf{b}(y)),\mathtt{lcs}(\mathbf{a}(x),y))\\
\mathtt{lcs}(\mathbf{b}(x),\mathbf{a}(y)) &\to \mathtt{max}(\mathtt{lcs}(x,\mathbf{a}(y)),\mathtt{lcs}(\mathbf{b}(x),y))
\end{aligned}$$

By putting $\mathtt{max} \prec_{\mathcal{F}} \mathtt{lcs}$, this is a MPO program. It admits a quasi-interpretation: $(\!|\mathbf{0}|\!) = (\!|\boldsymbol{\epsilon}|\!) = 1$, $(\!|\mathbf{S}|\!)(X) = (\!|\mathbf{i}|\!)(X) = (\!|\mathbf{j}|\!)(X) = X+1$, $(\!|\mathtt{max}|\!)(X,Y) = (\!|\mathtt{lcs}|\!)(X,Y) = \max(X,Y)$.

So the program computes a function in PTIME. Note that the explicit complexity of the program is exponential. The polynomial bound is obtained by the mean of memoization: the operational semantics is modified as shown in Figure 3. Every time a function call is computed, its result is stored in a cache and will be reused directly if the same call is needed another time.

ICAR is able to compute the value of a term with or without using the cache. It keeps a trace of the time and space used by the computation (*i.e.* the number of reduction steps and the maximum size of the cache and the environnement). The results for computing $\mathtt{lcs}(\mathbf{a}^n(\boldsymbol{\epsilon}),\mathbf{b}^n(\boldsymbol{\epsilon}))$ are:

| $n$ | `Call-by-value` | | `Memoization` | |
|---|---|---|---|---|
| | `time` | `space` | `time` | `space` |
| 0 | 4 | 1 | 4 | 1 |
| 1 | 17 | 2 | 17 | 5 |
| 2 | 63 | 2 | 48 | 10 |
| 3 | 219 | 2 | 97 | 17 |
| 4 | 771 | 2 | 164 | 26 |
| 5 | 2775 | 2 | 249 | 37 |
| 6 | 10169 | 2 | 352 | 50 |

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

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

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

$$\frac{\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \to \langle C_i, v_i \rangle \ \mathtt{f}(\vec{p}) \to r \in \mathcal{E} \ p_i\sigma' = v_i \ \mathcal{E}, \sigma' \vdash \langle C_n, r \rangle \to \langle C, v \rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \to \langle C \cup (\mathtt{f}(v_1, \cdots, v_n), v), v \rangle}$$

Figure 3: Call by Value interpreter with cache

# 4 Finding the precedence

This section describes the implementation of the two main difficulties of ICAR: finding the precedence for the termination orderings and finding the permutation for MPO. Both can be solved in polynomial time thanks to the restriction done upon the precedence.

**Lemma 10.** *Let $t = f(u_1, \cdots, u_n)$ and $s = g(v_1, \cdots, v_n)$ be two terms. If $g \approx_{\mathcal{F}} f$ implies $s \prec_{mpo} t$ then $g \prec_{\mathcal{F}} f$ implies $s \prec_{mpo} t$.*

*Proof.* If $t$ and $s$ are ordered when $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ then for all $i, 1 \leq i \leq n$, there exist a $j, 1 \leq j \leq n$ such that $v_i \preceq_{mpo} u_j$ (by definition of MPO). So for all $i$, $v_i \prec_{mpo} t$, so the two terms are ordered if $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. □

**Lemma 11.** *Let $t = f(u_1, \cdots, u_n)$ and $s = g(v_1, \cdots, v_n)$ be two terms. If $g \approx_{\mathcal{F}} f$ implies $s \prec_{lpo} t$ then $g \prec_{\mathcal{F}} f$ implies $s \prec_{lpo} t$.*

*Proof.* By definition of LPO, the hypothesis implies $v_i \prec_{lpo} t$. □

**Lemma 12.** *Let $t = f(u_1, \cdots, u_n)$ and $s = g(v_1, \cdots, v_n)$ be two terms. If $s \prec_{mpo} t$ or $s \prec_{lpo} t$, no symbol (function or constructor) in $s$ may have a precedence greater than the greatest symbol in $t$.*

*Proof.* Obvious since the symbol with the greatest precedence will lead to the greatest term. □

**Lemma 13.** *Let $l \to r$ be a rewriting rule such that $r \prec_{mpo} l$ or $r \prec_{lpo} l$. No symbol in $r$ may have a precedence greater than the head symbol of $l$.*

*Proof.* As $l$ is the left-hand-side of a rewriting rule, $l = \mathtt{f}(p_1, \cdots, p_n)$ where $p_i$ are patterns, that is terms build only over variables and constructors. Since constructors have a precedence smaller than any function symbol, f has the maximal precedence of $l$. So by previous lemma, it must also have the maximal precedence of $r$. $\square$

**Corollary 14.** *Finding the precedence is performed in polynomial time.*

*Proof.* By examining rewriting rules and by lemma 13, we obtain a set of constraints of the form $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$. Then, graph-reachability between any two symbol tells weather $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ or not. If there is both $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ and $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$, then there must be $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$. If there is only $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$, then lemma 10 and 11 tell that one won't looze anything by choosing $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. So the precedence is found in polynomial time. $\square$

# References

[1] Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. On lexicographic termination ordering with space bound certifications. Tech. rep., Loria, Mar 2001. accepted to Andre Ershov Fourth International Conference (PSI01).

[2] Jones, N. The expressive power of higher order types or, life without cons. to appear, 2000.

[3] Marion, J.-Y. Analysing the implicit complexity of programs. *Information and Computation* (2000). to appear.

[4] Marion, J.-Y., and Moyen, J.-Y. Efficient first order functional program interpreter with time bound certifications. In *LPAR* (Nov 2000), vol. 1955 of *Lecture Notes in Computer Science*, Springer, pp. 25–42.