Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html pages

System Presentation: An Analyser of Rewriting Systems Complexity

J.-Y. Moyen [1]

Loria, Calligramme project,
B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France

Abstract

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

1 The Analyser ICAR

We consider term rewriting systems built over three distinct sets: function symbols (defined symbols), constructors, and variables. The function symbols and constructors are ordered by a precedence $\prec_{\mathcal{F}}$, constructors are considered as the smallest 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 [5,7,2] for complexity analysis. Program transformation by means of memoization is based on Jones work [4].

- **Termination** is checked using a 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 interests 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 operational semantics) to effectively achieve this bound.

[1] Email: moyen@loria.fr

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

ICAR is written in Objective Caml. One may download Objective Caml from http://caml.inria.fr and ICAR from http://www.loria.fr/ moyen/.

2 Programs and Semantics

2.1 Rewriting Systems

Definition 2.1 The sets of terms, patterns and function rules are defined in the following way:

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

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$[2]. The size $|t|$ of a term t is the number of symbols in t.

Definition 2.2 A program is a set $\mathcal{E}$ of $\mathcal{D}$-rules such that for each rule $\mathtt{f}(p_1, \cdots, p_n) \to t$ of $\mathcal{E}$, each variable in t appears also in some pattern p_i.

2.2 Semantics

The signature $\mathcal{C} \cup \mathcal{F}$ and the set $\mathcal{E}$ of rules induce a rewrite system which brings us the operational semantics. Recall briefly some vocabulary of rewriting theories. For further details, one might consult Dershowitz and Jouannaud's survey [3] from which we take the notation. 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 normal form, *i.e.* no other rule may be applied.

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 2.3 Let *main* be a confluent program and $\mathtt{main} \in \mathcal{F}$ be a function symbol. The function computed by *main* is the partial function $[\![\mathtt{main}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ where n is the arity of $\mathtt{main}$ which is defined as follows. For

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

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

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

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

Fig. 1. Multiset Path Ordering

all $u_i \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{main}]\!](u_1, \cdots, u_n) = v$ iff $\mathtt{main}(u_1, \cdots, u_n) \xrightarrow{!} v$ 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, $[\![\mathtt{main}]\!](u_1, \cdots, u_n)$ is undefined.

3 Termination Ordering and Quasi-Interpretations

3.1 Termination Orderings

Definition 3.1 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 π such that:

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

Definition 3.2 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 3.3 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 3.4 The *Lexicographic Path Ordering* (LPO) is defined in the rules of Figure 2.

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

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

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

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

Fig. 2. Lexicographic Path Ordering

3.2 Quasi-Interpretation

Definition 3.5 A *quasi-interpretation* of a symbol f of arity n is a function $(\!|f|\!) : \mathbb{N}^n \mapsto \mathbb{N}$ 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 recursively 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 $l \to r$, $(\!|r|\!) \leq (\!|l|\!)$. Quasi-interpretations are not sufficient to prove program termination. Indeed the one-rule program $\mathtt{f}(x) \to \mathtt{f}(x)$ admits a polynomial quasi-interpretation but doesn't terminate.

Remark 3.6 If one replace all inequalities in the previous definition by strict inequalities, one will obtain the definition of polynomial interpretations. Interpretations have been widely used to prove program termination. Recently, Bonfante, Cichon, Marion and Touzet in [1] have shown that polynomial interpretations by themselves may also give a bound on program complexity.

But interpretations are more restrictive than quasi-interpretations, and some programs actually admit a quasi-interpretation but no interpretation.

3.3 Theorems

Theorem 3.7 *The set of functions computable by MPO-programs admitting quasi-interpretations is* exactly Ptime, *the set of functions computable in polynomial time.*

Proof. See [7]. □

Theorem 3.8 *The set of functions computable by LPO-programs admitting quasi-interpretations is* exactly PSPACE*, the set of functions computable in polynomial space.*

Proof. See [2] □

4 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. Fortunately ICAR uses a restriction of the usual orderings: a constructor always has a precedence smaller than any function symbol.

Claim 4.1 *With this restriction, the precedence can be found in quadratic time with respect to program size. 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 5.1 □

The main difficulty lies in the second part. Indeed, it is not known if quasi-interpretations are decidable. The similar problem with strict inequalities (*i.e.* finding polynomial interpretations) seems to be undecidable. Fortunately, quasi-interpretations are quite easy to find because an upper bound on the program denotation turns out to be a good candidate. So, even if finding one is a hard task for a computer, it's 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 delegate this job to such a program. Currently, ICAR uses Maple as a quasi-interpretation checker. Maple may be unable to check some inequalities (especially those using a lot of maxs). As far as I know, there isn't any software able to treat them properly, but one is under developement by Fabrice Rouiller at LIP6 (Paris) and should be available by the end of the year.

In the current implementation, ICAR returns the quasi-interpretations that Maple was unable to solve and hopes that the user will be less clumsy.

Example 4.2

(i) 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:

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

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

(ii) One may compute the length of the longest common subsequence of two strings 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 an MPO-program. It admits a quasi-interpretation: $(\!|\mathbf{0}|\!) = (\!|\boldsymbol{\epsilon}|\!) = 1$, $(\!|\mathbf{S}|\!)(X) = (\!|\mathbf{a}|\!)(X) = (\!|\mathbf{b}|\!)(X) = X + 1$, $(\!|\mathtt{max}|\!)(X, Y) = (\!|\mathtt{lcs}|\!)(X, Y) = \max(X, Y)$.

So the program computes a function in PTIME. Note that the explicit complexity of the program is exponential. The polynomial bound is obtained by means 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

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

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

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

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

Fig. 3. Call by Value interpreter with cache

(*i.e.* the number of reduction steps and the maximum size of the cache and the environment). 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

Time measurement is the number of reduction steps needed to reach normal form and space measurement is the size of the environment and the cache, *i.e.* the number of objects stored into the environment plus the number of objects stored in the cache. These measurements are not very accurate because the real time needed to perform a reduction step and the size of a term depend on the term. But in both cases, the increment is polynomial in the size of the term, so the polynomial bounds are not broken and the measurements give a not so bad idea of what happens.

Currently, every time a function call is computed, its result is stored in the cache. So, there are a lot of unused things in the cache, in the sense that some computations are only done once but nevertheless stored: this part of the cache won't be used. But the rewriting system is ordered by MPO, and this gives a lot of informations about it. So, knowing it, there is actually a way to minimize the cache, as described in [6]

5 More on the Implementation

5.1 Finding the Precedence in Polynomial Time

Lemma 5.1 *Let $t = \mathtt{f}(u_1, \cdots, u_n)$ and $s = \mathtt{g}(v_1, \cdots, v_n)$ be two terms. If $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ implies $s \prec_{mpo} t$ then $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$ 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 exists 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 5.2 *Let $t = \mathtt{f}(u_1, \cdots, u_n)$ and $s = \mathtt{g}(v_1, \cdots, v_n)$ be two terms. If $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ implies $s \prec_{lpo} t$ then $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$ implies $s \prec_{lpo} t$.*

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

Lemma 5.3 *Let $t = \mathtt{f}(u_1, \cdots, u_n)$ and $s = \mathtt{g}(v_1, \cdots, v_n)$ 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 5.4 *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 the previous lemma, it must also have the maximal precedence of r. □

Corollary 5.5 *Finding the precedence is performed in polynomial time.*

Proof. By examining rewriting rules and by lemma 5.4, we obtain a set of constraints of the form $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$. Then, graph-reachability between any two symbol tells whether $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ or not. If there are 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 5.1 and 5.2 tell that one won't lose anything by choosing $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. So the precedence is found in polynomial time. □

So, this leads to the following algorithm for finding precedence:

(i) For each function symbol $\mathtt{f}$, create three empty sets $A_{\mathtt{f}}$, $B_{\mathtt{f}}$ and $C_{\mathtt{f}}$.
(ii) For each rule $\mathtt{f}(p_1, \cdots, p_n) \to r$ and for each function symbol $\mathtt{g} \in r$, add $\mathtt{g}$ in $A_{\mathtt{f}}$.
(iii) For each function symbol $\mathtt{f}$, do:
  (a) Add $\mathtt{f}$ to $C_{\mathtt{f}}$.
  (b) $B_{\mathtt{f}} \leftarrow A_{\mathtt{f}}$
(iv) While $\exists\mathtt{f}$ such that $\exists\mathtt{g} \in B_{\mathtt{f}}$ do
  (a) Add $\mathtt{g}$ to $C_{\mathtt{f}}$.
  (b) $B_{\mathtt{f}} \leftarrow B_{\mathtt{f}} \bigcup A_{\mathtt{g}} \backslash \mathtt{g}$.
(v) If $\mathtt{f} \in C_{\mathtt{g}}$ and $\mathtt{g} \in C_{\mathtt{f}}$, then $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$. If $\mathtt{f} \in C_{\mathtt{g}}$ and $\mathtt{g} \notin C_{\mathtt{f}}$, then $\mathtt{f} \prec_{\mathcal{F}} \mathtt{g}$.

The first step of the algorithm scans the rules and builds a first set of constraints of the form $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$. It is done in time linear in the size of the program. The loop builds the transitive-reflexive closure of the constraints previously obtained by means of graph reachability. It is done in time quadratic in the number of function symbols.

The set $A_{\mathtt{f}}$ contains the immediate neighbors of $\mathtt{f}$. $B_{\mathtt{f}}$ contains the nodes reachable from $\mathtt{f}$ but yet unprocessed and $C_{\mathtt{f}}$ contains the nodes reachable from $\mathtt{f}$ and already processed.

5.2 Orderings and Quasi-Interpretations

Once the precedence has been found, checking termination by either ordering is quite easy. One just needs to compare the head symbols of the two terms and follow the definitions of Figures 1 and 2.

As previously said, finding a polynomial quasi-interpretation is probably an undecidable problem. But it appears that the semantics of a function is a good candidate (*e.g.* the function add, computing the addition of two numbers, admits addition as a quasi-interpretation). Of course, finding the semantics automatically is also undecidable. But when one writes a program, one is suppose to know what the program will compute, so one can give to ICAR something very likely to be a quasi-interpretation.

6 A Simple Session

First, load a program and print it.

```
>load mult
>print
add(Z,x0) -> x0
add(S(x0),x1) -> S(add(x0,x1))
mult(Z,x0) -> Z
mult(S(x0),x1) -> add(x1,mult(x0,x1))

[S](x0)=(x0)+(1)
```

```
[Z]()=1
[mult](x0,x1)=(x0)*(x1)
[add](x0,x1)=(x0)+(x1)
```

The program contains both the rewriting rules and the quasi-interpretations. ICAR can then check its complexity.

```
>check
The program terminates by MPO.
Quasi-interpretation are OK.
The function is Ptime computable.
```

One can load a new program, check its complexity, and compute the value of a term using different operational semantics.

```
>load lcs
>print
maxi(Z,x0) -> x0
maxi(x0,Z) -> x0
maxi(S(x0),S(x1)) -> S(maxi(x0,x1))
lcs(E,x0) -> Z
lcs(x0,E) -> Z
lcs(A(x0),A(x1)) -> S(lcs(x0,x1))
lcs(B(x0),B(x1)) -> S(lcs(x0,x1))
lcs(A(x0),B(x1)) -> maxi(lcs(A(x0),x1),lcs(x0,B(x1)))
lcs(B(x0),A(x1)) -> maxi(lcs(B(x0),x1),lcs(x0,A(x1)))

[B](x0)=(x0)+(1)
[A](x0)=(x0)+(1)
[E]()=1
[S](x0)=(x0)+(1)
[Z]()=1
[lcs](x0,x1)=max(x1,x0)
[maxi](x0,x1)=max(x1,x0)

>check
The program terminates by MPO.
Quasi-interpretation are OK.
The function is Ptime computable.

>setcbv
>getsem
Call by value
>eval lcs(A(A(A(E))),B(B(B(E))))
Time usage : 219
Space usage: 2
Result     : Z
```

```
>setmem
>getsem
Memoization
>eval lcs(A(A(A(E))),B(B(B(E))))
Time usage : 97
Space usage: 17
Result     : Z
```

`setcbv` and `setmem` toggle either the call-by-value semantics or the memoization. `getsem` returns the current semantics.

The time and space measurment are the number of reduction steps needed, and the number of terms stored in the environment and in the cache.

```
>load exp
>print
d(Z) -> Z
d(S(x0)) -> S(S(d(x0)))
exp(Z) -> S(Z)
exp(S(x0)) -> d(exp(x0))

[S](x0)=(x0)+(1)
[Z]()=1
[exp](x0)=x0
[d](x0)=(2)*(x0)

>check
The program terminates by MPO.
The equation 3 is not decreasing for the Quasi-interpretation.
The function is Primitive Recursive.


>load add_error
>print
add(Z,x0) -> x0
add(S(x0),x1) -> S(add(x0,x1))

[S](x0)=(x0)+(1)
[Z]()=1
[add](x0,x1)=max(x1,x0)

>check
The program terminates by MPO.
Cannot check equation 2: ((max((x1)+(1),x0))-(max(x0,x1)))-(1).
The function is Primitive Recursive.
```

Maple may be able to discover some trouble with the quasi-interpretations. Currently, nothing is done to try to correct it, the user has to find another quasi-interpretation, and retry it.

The returned result is either the first equation that is strictly increasing with respect to the quasi-interpretation (*i.e.* the interpretation of the left hand side is strictly smaller than the interpretation of the right hand side) or the polynomials whose sign Maple was unable to discover (in this case, because the sign is not constant).

```
>load qbf
>print
or(TRUE,x0) -> TRUE
or(FALSE,x0) -> x0
and(TRUE,x0) -> x0
and(FALSE,x0) -> FALSE
eq(Z,Z) -> TRUE
eq(S(x0),S(x1)) -> eq(x0,x1)
eq(Z,x0) -> FALSE
eq(x0,Z) -> FALSE
mem(x0,NIL) -> FALSE
mem(x0,CONS(x1,x2)) -> or(eq(x0,x1),mem(x0,x2))
qbf(VAR(x0),x1) -> mem(x0,x1)
qbf(OR(x0,x1),x2) -> or(qbf(x0,x2),qbf(x1,x2))
qbf(AND(x0,x1),x2) -> and(qbf(x0,x2),qbf(x1,x2))
qbf(EXISTS(x0,x1),x2) -> or(qbf(x1,x2),qbf(x1,CONS(x0,x2)))
qbf(FORALL(x0,x1),x2) -> and(qbf(x1,x2),qbf(x1,CONS(x0,x2)))

[FORALL](x0,x1)=((x0)+(x1))+(1)
[EXISTS](x0,x1)=((x0)+(x1))+(1)
[AND](x0,x1)=((x0)+(x1))+(1)
[OR](x0,x1)=((x0)+(x1))+(1)
[VAR](x0)=(x0)+(1)
[CONS](x0,x1)=((x0)+(x1))+(1)
[NIL]()=1
[S](x0)=(x0)+(1)
[Z]()=1
[FALSE]()=1
[TRUE]()=1
[qbf](x0,x1)=(x0)+(x1)
[mem](x0,x1)=max(x1,x0)
[eq](x0,x1)=max(x1,x0)
[and](x0,x1)=max(x1,x0)
[or](x0,x1)=max(x1,x0)

>check
```

```
The program doesn't terminates by MPO.
The program terminates by LPO.
Cannot check equation 10:
        (max(((x1)+(x2))+(1),x0))-(max(x0,max(x2,x1))).
The function is Multiple recursive.
```

On some more tricky examples, Maple may be unable to find the sign of one or more polynomials, even if it is constant. This could probably be fixed with a more clever use of Maple than the one currently implemented.

7 Future of ICAR

The idea is to plug ICAR into ELAN[3], which is a quite big system developed at Loria to deal with term rewriting systems. Having a bound on the complexity of some of the systems will be very usefull for the ELAN-people. During spring and summer 2001, Mitch Harris rewrote ICAR in TOM, the (futur) parser of ELAN terms.

References

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

[2] Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. On lexicographic termination ordering with space bound certifications. In *PSI* (Jul 2001), Lecture Notes in Computer Science, Springer.

[3] Dershowitz, N., and Jouannaud, J.-P. *Handbook of Theoretical Computer Science vol.B.* Elsevier Science Publishers B. V. (NorthHolland), 1990, ch. Rewrite systems, pp. 243–320.

[4] Jones, N. The expressive power of higher order types or, life without cons. *Journal of Functional Programming 11*, 1 (2000), 55–94.

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

[6] Marion, J.-Y. Complexité implicite des calculs, de la thorie la pratique, 2000. Habilitation.

[7] 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.

[3] http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/