*URL:* [*http://www.elsevier.nl/locate/entcs/volume59.html*](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 Vand uvre-l es-Nancy Cedex, France*

*Abstract*

*This paper describes ICAR, a program which analyses the implicit complexity of*

*rst 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 (de ned symbols), constructors, and variables. The function symbols and constructors are ordered by a precedence F , constructors are considered as the smallest elements of the precedence. A program is a set of rewriting rules.

*ICAR (Implicit Complexity AnalyseR) rst checks the termination of the* given rewriting system and then tries to nd 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 or-* dering 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 com- plexity of the program. This kind of complexity analysis was dubbed implicit. So ICAR also gives a way (i.e. a new operational semantics) to e ectively achieve this bound.

*1 Email:* [*moyen@loria.fr*](mailto:moyen@loria.fr)

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license*](http://creativecommons.org/licenses/by-nc-nd/3.0/)*.*

*One may then run the program using di erent 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](http://caml.inria.fr/) and ICAR from <http://www.loria.fr/> moyen/.

# *2 Programs and Semantics*

*2.1 Rewriting Systems*

*De nition 2.1 The sets of terms, patterns and function rules are de ned in* the following way:

*(Constructor terms) T (C) 3 u ::=c j c(u1; ; un)*

*(Ground terms) T (C; F) 3 s ::=c j c(s1; ; sn) j f(s1; ; sn) (terms) T (C; F; X ) 3 t ::=c j x j c(t1; ; tn) j f(t1; ; tn) (patterns) P 3 p ::=c j x j c(p1; ; pn)*

*(rules) D 3 d ::=f(p1; ; pn) ! t*

*where x 2 X , f 2 F, and c 2 C 2 . The size jtj of a term t is the number of* symbols in t.

*De nition 2.2 A program is a set E of D-rules such that for each rule* f(p1; ; pn) ! t of E , each variable in t appears also in some pattern pi.

*2.2 Semantics*

*The signature C[F and the set E of rules induce a rewrite system which brings* us the operational semantics. Recall brie y 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 ! induced by a program main is de ned as follows t ! s if s is obtained from t by

*applying one of the rules of E . The relation ! is the re exive-transitive*

*!*

*closure of !. Lastly, t!s means that t!s and s is in normal form, i.e. no*

*other rule may be applied.*

*We now give the semantics of con uent programs, that is programs for* which the associated rewrite system is con uent. The domain of interpretation is the constructor algebra T (C).

*De nition 2.3 Let main be a con uent program and main 2 F be a function* symbol. The function computed by main is the partial function JmainK : T (C)n ! T (C) where n is the arity of main which is de ned as follows. For

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

*s mpo ti*

*s mpo f (::: ; ti;::: )*

*f 2 F S C*

*si mpo g(t1; ; tn) f F g* f (s1; ; sm) mpo g(t1; ; tn)

*g 2 F;f 2 F S C*

*fs1; ; sng m ft1; ; tng f F g*

*mpo*

*f; g 2 F*

*g(s1; ; sn) mpo f(t1; ; tn)*

*all u*

*Fig. 1. Multiset Path Ordering*

*2 T (C); JmainK(u ; ;u ) = v i main(u ; ;u )!!*

*v with v 2 T (C).*

*i 1 n 1 n*

*Note that due to the form of the rules a constructor term is a normal form ;*

*as the program is con uent, it is uniquely de ned. Otherwise, that is if there* is no such normal form, JmainK(u1; ; un) is unde ned.

# *3 Termination Ordering and Quasi-Interpretations*

*3.1 Termination Orderings*

*De nition 3.1 Let be an ordering over terms. The multiset extension m* of is de ned as follow:

*M = fm1; ; mkg m fn1; ; nkg = N if and only if M 6= N and there* exists a permutation such that:

*There exists j such that mj n (j)*

*For all i, mi n (i)*

*De nition 3.2 The Multiset Path Ordering (MPO) is de ned in the rules of* Figure 1.

*A MPO-program is a program such that for each rule l ! r, r mpo l.*

*De nition 3.3 Let be an ordering over terms. The lexicographic extension*

*l of is recursively de ned as follow:*

*(t1; ; tn) l (s1; ; sm) if and only if t1 s1 or t1 = s1 and (t2;::: ; tn) l* (s2;::: ; sm).

*De nition 3.4 The Lexicographic Path Ordering (LPO) is de ned in the* rules of Figure 2.

*A LPO-program is a program such that for each rule l ! r, r lpo l.*

*s lpo ti*

*s lpo f(::: ; ti;::: )*

*si lpo f(t1; ; tn) g F f* g(s1; ; sm) lpo f(t1; ; tn)

*(s1; ; sn) l*

*lpo*

*(t1; ; tn) f F g sj lpo f(t1; ; tn)*

*g(s1; ; sn) lpo f(t1; ; tn)*

*Fig. 2. Lexicographic Path Ordering*

*3.2 Quasi-Interpretation*

*De nition 3.5 A quasi-interpretation of a symbol f of arity n is a function* Lf M : N n 7! N such that:

*Lf M is bounded by a polynomial*

*Lf M is (non strictly) increasing*

*Lf M(X1; ; Xn) Xi for all i n.*

*LcM(X1; ;X ) = Pn*

*X + for all constructors c where > 0 is a*

*n*

*constant.*

*i=1* *i*

*Quasi-interpretations are extended recursively to terms as usual:* Lf (t1; ; tn)M = Lf M(Lt1M;::: ; LtnM)

*A program admits a quasi-interpretation if for each rule l ! r, LrM LlM.* Quasi-interpretations are not suÆcient to prove program termination. Indeed the one-rule program f(x) ! f(x) admits a polynomial quasi-interpretation but doesn't terminate.

*Remark 3.6 If one replace all inequalities in the previous de nition by strict* inequalities, one will obtain the de nition of polynomial interpretations. In- terpretations have been widely used to prove program termination. Recently, Bonfante, Cichon, Marion and Touzet in [1] have shown that polynomial in- terpretations 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 poly- nomial time.

*Proof. See [7].* *2*

*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]* *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 2c k where k is the maximal arity of a function symbol and c is a constant.

*Proof. See section 5.1* *2*

*The main diÆculty lies in the second part. Indeed, it is not known if quasi-* interpretations are decidable. The similar problem with strict inequalities (i.e.

*nding polynomial interpretations) seems to be undecidable. Fortunately,* quasi-interpretations are quite easy to nd because an upper bound on the program denotation turns out to be a good candidate. So, even if nding 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.

*add(0; y) ! y*

*add(S(x); y) ! S(add(x; y)) mult(0; y) ! 0*

*mult(S(x); y) ! add(y; mult(x; y))*

*It terminates either by MPO or LPO by putting add F mult and admits* a quasi-interpretation: LaddM(X; Y ) = X + Y , LmultM(X; Y ) = X Y , L0M = 1, LSM(X) = X + 1. Let's verify the quasi-interpretaion for the last rule:

*Lmult(S(x); y)M = LmultM(LS(x)M;Y ) = (X + 1) Y*

*Ladd(y; mult(x; y))M = Y + Lmult(x; y)M = Y + X Y*

*(ii) One may compute the length of the longest common subsequence of two* strings as follow:

*max(n; 0) ! n*

*max(0; m) ! m*

*max(S(n); S(m)) ! S(max(n; m))*

*lcs(x; ) ! 0*

*lcs( ; y) ! 0*

*lcs(a(x); a(y)) ! S(lcs(x; y))*

*lcs(b(x); b(y)) ! S(lcs(x; y))*

*lcs(a(x); b(y)) ! max(lcs(x; b(y)); lcs(a(x); y))*

*lcs(b(x); a(y)) ! max(lcs(x; a(y)); lcs(b(x); y))*

*By putting max F lcs, this is an MPO-program. It admits a quasi-* interpretation: L0M = L M = 1, LSM(X) = LaM(X) = LbM(X) = X + 1, LmaxM(X; Y ) = LlcsM(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 ob- tained by means of memoization: the operational semantics is modi ed 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*

*(x) = v*

*E ; ` hC; xi ! hC; vi*

*c 2 C E ; ` hCi 1; tii ! hCi; vii*

*E ; ` hC0; c(t1; ; tn)i ! hCn; c(v1; ; vn)i*

*f 2 F E ; ` hCi 1; tii ! hCi; vii (f(v1; ; vn); v) 2 Cn* E ; ` hC0; f(t1; ; tn)i ! hCn; vi

*E ; ` hC*

*;t i ! hC ;v i f(p) ! r 2 E p 0 = v*

*E ; 0 ` hC*

*; ri ! hC; vi*

*i 1 i i* *i*

*i i n*

*E ; ` hC0; f(t1; ; tn)i ! hC [ (f(v1; ; vn); v); vi*

*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 lcs(an( ); bn( )) 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 = f(u1; ; un) and s = g(v1; ; vn) be two terms. If* g F f implies s mpo t then g F f implies s mpo t.

*Proof. If t and s are ordered when g F f then for all i; 1 i n, there* exists a j; 1 j n such that vi mpo uj (by de nition of MPO). So for all i, vi mpo t, so the two terms are ordered if g F f. *2*

*Lemma 5.2 Let t = f(u1; ; un) and s = g(v1; ; vn) be two terms. If* g F f implies s lpo t then g F f implies s lpo t.

*Proof. By de nition of LPO, the hypothesis implies vi lpo t.* *2*

*Lemma 5.3 Let t = f(u1; ; un) and s = g(v1; ; vn) be two terms. If* s mpo t or s 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. *2*

*Lemma 5.4 Let l ! r be a rewriting rule such that r mpo l or r 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 = f(p1; ; pn) where* pi 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. *2*

*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 f F g. Then, graph-reachability between any two symbol tells whether f F g or not. If there are both f F g and g F f, then there must be f F g. If there is only g F f, then lemma 5.1 and *5.2*

*tell that one won't lose anything by choosing g F f. So the precedence is*

*found in polynomial time.* *2*

*So, this leads to the following algorithm for nding precedence:*

*(i) For each function symbol f, create three empty sets Af, Bf and Cf.*

*(ii) For each rule f(p1; ; pn) ! r and for each function symbol g 2 r, add* g in Af.

*(iii) For each function symbol f, do:*

*(a) Add f to Cf.*

*(b) Bf Af*

*(iv) While 9f such that 9g 2 Bf do*

*(a) Add g to Cf.*

*(b) Bf Bf S Agng.*

*(v) If f 2 Cg and g 2 Cf, then f F g. If f 2 Cg and g 2= Cf, then f F g.* The rst step of the algorithm scans the rules and builds a rst set of con-

*straints of the form f F g. It is done in time linear in the size of the program.*

*The loop builds the transitive-re exive 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 Af contains the immediate neighbors of f. Bf contains the nodes* reachable from f but yet unprocessed and Cf contains the nodes reachable from 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 de nitions of Figures 1 and 2.

*As previously said, nding 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, nding 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 di erent 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 memoiza-* tion. 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 nd another quasi-interpretation, and retry it.

*The returned result is either the rst 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 nd the sign of one* or more polynomials, even if it is constant. This could probably be xed 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 devel-* oped 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 certi cations. 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. Complexit e implicite des calculs, de la thorie la pratique, 2000. Habilitation.*

*[7] Marion, J.-Y., and Moyen, J.-Y. EÆcient rst order functional program interpreter with time bound certi cations. 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/*](http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/)