Abstract Complexity of Prolog Based on WAM

Alexander Ja. Dikovsky

Russian Academy Inst. for Applied Mathematics,
4, Miusskaja sq., Moscow, Russia 125047
Email: dikovsky@applmat.msk.su
Fax: (7-095-)972-0737

Abstract. Simple definitions of time and space complexity measures of Prolog programs based on an abstraction of Warren Abstract Machine instructions set is suggested. Its effect is discussed on practical measuring time and space consumption, complexity classifications and optimization of programs.

1 INTRODUCTION

Proper choice of adequate time and space complexity criteria for computations in Prolog needs thorough analysis. If we try to conceive criteria based on the standard resolution semantics we would only come to somewhat like logical inferences used in practice for measuring time and to some total weight of terms used in a program for measuring space.

The notion of "logical inference" (LI) is somewhat fuzzy. Sometimes LI corresponds to one unfold (or resolution) step, and sometimes - to one unification of two terms. These are in fact of the same order. However, they differ for asymptotic time bounds. Moreover, for measuring time of highly nondeterministic programs it is convenient to distinguish between unfold steps and backtrack steps (BT), thus measuring the total time in Lk and BTs. In any case \( \text{time}_{BT}\text{(computation)} < \text{time}_{LI}\text{(computation)} \). However, for deterministic programs \( \text{time}_{BT} = 0 \), whereas for backtrack-choice programs it is close to \( \text{time}_{LI} \).

Much more complex is the problem of choice of adequate space complexity measures for Prolog. Term-weight measures (cf.[4]) reflect that aspect of space consumption which pertains to generating terms. But it does not cope with space needed for control. For example, consider the following logically natural program which counts the number of non-list items in complex lists.

Example 1 % nlin(+Complex_list,-Items_number).
nlin([ ],0).
nlin([E|T],N) :-
  nlin(E,N1),
  nlin(T,N2),
  N is N1+N2.
nlin(_,1).
All terms unified in its computations are garbage. However, it wastes a lot of space, and can overwhelm Prolog workspace for complex enough lists. Standard semantics of Prolog founded on SLD-resolution rule [8] gives no terms to account for workspace size. This size depends on implementation. After long evolution de facto standard of compiling Prolog has been formed embodied in Warren Abstract Machine (WAM) instructions set [11,1]. Almost all professional interpreters and compilers of Prolog use standard stacks: local stack(s) for activation frames and choice points, trail for backtrackable variables and global stack (or heap) for lists and structures [11,1,9]. So, we might think of abstract Prolog interpreter as of a sort of multi-stack automata, and define Prolog workspace as the size of its stacks. This, however would be too straightforward because the bulk of WAM instructions implement various recursion optimization rules: tail recursion optimization [3, 10], last call or activation frame optimization [11,1], arguments indexing [11,1], garbage collection [2], and so on. In presence of these rules abstract interpreter becomes too overburdened by technical details and this is the main difficulty in space complexity abstraction. Nevertheless space complexity measures definitely must reflect the effect of WAM optimizations so that to be practically feasible. WAM optimizations indirectly regulate the style of practical programming. For example, more practical version of the nlin/2 predicate above should be somewhat like this:

Example 2  
% nlin(+Complex_list,-Items_number).
\[
\text{nlin}((L,N) :-
\text{nlin}_t(L,[ ],0,N).
\text{nlin}_t([ ],[ ],N,N) :- !.
\text{nlin}_t([ ],[E|T],N,R) :- !,
\text{nlin}_t(E,T,N,R).
\text{nlin}_t([E|T],S,N,R) :- !,
\text{nlin}_t(E,[T|S],N,R).
\text{nlin}_t([ ],L,N1,R) :-
N1 is N+1,
\text{nlin}_t([ ],L,N1,R).
\]

And the measures should indicate the reason for which this program needs constant workspace.

2 ASM-COMPLEXITY

Of course it would be contradictio in adjecto if so that to write or estimate programs in logic programming language we should know details of its implementation. Instead we should use very simple abstract model of standard interpreter sufficient for adequate measuring workspace. We have offered such a model called abstract stack machine (ASM) in [5,6] and sketch it here. It is founded on the
notion of a derivation tree, representing a successful branch of SLD-tree. We cite here several basic definitions from [6].

**Definition 1** A logical procedure is a set \( lp/n \) of definitions of predicates with one distinguished predicate \( main/n \). An instance of logical procedure \( lp/n \) or a program is the set of definitions \( lp/n \) and a query

\[ ? - main(t_1, ..., t_n). \]

We denote it \( lp(t_1, ..., t_n) \).

**Definition 2** A pair \( t = < T, f > \) where \( T \) is a completely ordered tree and \( f \) is a focus of \( T \) is called a focused tree; \( f \) is called a label of \( T \). Let \( L \) be some set, \( t = < T, f > \) be a focused tree and \( l \) be a function from \( T \) to \( L \). Then \( s = < T, f, l > \) is called a labelled focused tree, \( l \) is called a labelling and for \( v \) in \( T \) \( l(v) \) is called a label of \( v \). A labelled focused tree is a state if any node to the right of its focus is a leaf. Let \( pr = lp(W) \) be a program, \( s = < T, f, l > \) is a state of \( pr \) if for all \( v < f \) \( l(v) = (a_v, i_v, u_v) \) and for all \( v \geq f \) \( l(v) = (a_v, i_v) \), where all \( a_v \) are atoms in \( pr \), all \( i_v \) are nonnegative integers and all \( u_v \) are some substitutions of terms for variables in \( a_v \). Labels of nodes of states are called subgoals, those to the left of the focus - the accessible ones, all others - resolvent. The subgoal labelling the root is called a query subgoal; variables in this subgoal (if any) are called query variables. To simplify notation we do not distinguish nodes of states and occurrences of subgoals labelling them.

![Diagram](image.png)

**Fig. 1.** A state in the computation of \( nlin([p_1, p_2], N). \)
The labelled focused tree in the figure 1 is a state in the computation of the program \textit{nlin}([(\textit{p}_1, \textit{p}_2), \textit{p}_3], \textit{N}) which is an instance of the logical procedure \textit{nlin}/2 defined in example 1 (for accuracy we should substitute \textit{main}/2 for \textit{nlin}/2 in this definition). Underlined is the focus subgoal. \textit{u}_1, \textit{u}_2, \textit{u}_3 and \textit{u}_4 are the MGUs found on preceeding steps.

A program state can be represented naturally by three stacks.

\textbf{Definition 3} Let \textit{s} = <\textit{T}, \textit{f}, \textit{l}> be a state of a program \textit{lp}(\textit{W}) and

\[(a_1, i_1, u_1), ..., (a_k, i_k, u_k), (a_{k+1}, i_{k+1}), ..., (a_n, i_n)\]

be the sequence of all its subgoals in increasing order. Then the sequence \((a_1, i_1), ..., (a_k, i_k)\) is called an accessible subgoals stack (AS), the sequence \(u_1, ..., u_k\) is called a unifiers stack (US) and the sequence \((a_{k+1}, i_{k+1}), ..., (a_n, i_n)\) is called a resolvent stack (RS). \((a_k, i_k)\) and \(u_k\) are the top elements of AS and US respectively. The focus subgoal \((a_{k+1}, i_{k+1})\) is the top element of RS. The composition \textit{con}(\textit{s}) = \textit{u}_1 \circ ... \circ \textit{u}_k\) of all substitutions in US is called a context of the state \textit{s}.

It is clear that states and stacks defined on them determine each other uniquely, so transformations of states can be described also in terms of transformations of their stacks. Transitions from states to states in AS-machines are defined naturally in terms of two operators on stacks of states: \textit{unfold} and \textit{backtrack}. \textit{unfold} chooses the next applied clause \textit{head} : = \textit{body}. and the corresponding MGU \textit{u}, pops the focus subgoal unified with head from RS, pushes head on AS, the body (if any) on RS and \textit{u} on US. \textit{backtrack} performs the reverse transformation of stacks (saving used clause number). We refer to [6] for details. So we come to the natural definition of a computation.

\textbf{Definition 4} The starting state of AS-machine under the program \textit{pr} = \textit{lp}(\textit{w}_1, ..., \textit{w}_m) is the state \textit{s}_0 in which AS and US are empty and RS contains only the query subgoal \((\textit{main}(\textit{w}_1, ..., \textit{w}_m), 0)\). The computation of \textit{pr} is the sequence \textit{comp}(\textit{pr}) = \((\textit{s}_0, \textit{s}_1, ..., \textit{s}_n)\) of states in which

\[\textit{s}_{i+1} = \begin{cases} \textit{unfold}(\textit{s}_i), & \text{if it applies;} \\ \textit{backtrack}(\textit{s}_i), & \text{otherwise.} \end{cases}\]

for each \(i\).

An accessible subgoal \((\textit{p}(t_1, ..., t_k), d, u)\) in \textit{s}_i with \textit{p}/\textit{k} defined in \textit{lp}/\textit{m} is called deterministic if \textit{d} equals to the number of clauses in the definition of \textit{p}/\textit{k}. Otherwise this subgoal is called a choice point.

If \textit{comp}(\textit{pr}) = \((\textit{s}_0, ..., \textit{s}_n)\), \textit{s}_0 = \textit{unfold}(\textit{s}_{n-1}) and RS is empty in \textit{s}_n then \textit{comp}(\textit{pr}) is successful. A result of a successful computation of \textit{pr} is the substitution \textit{res}(\textit{pr}) which is the restriction of \textit{con}(\textit{s}_n) to query variables. \textit{comp}(\textit{pr}) = \((\textit{s}_0, ..., \textit{s}_n)\) is unsuccessful if \textit{s}_n = \textit{backtrack}(\textit{s}_{n-1}) and AS is empty in \textit{s}_n.

So as to define complete AS-machine semantics of a dialect of Prolog we should define for each built-in \textit{p}/\textit{n} values \textit{unfold}(\textit{s}) and \textit{backtrack}(\textit{s}) on those
states \( s \) where a subgoal \( (p(\bar{U}), d) \) is on the top of \( RS \) or a subgoal \( (p(\bar{U}), d, u) \) is on the top of \( AS \) respectively. For example we define completely kernel Prolog, i.e. Prolog with the only one built-in operator \( 	ext{cut} !/0 \) if the following definition is added.

If \( g = (!, 0) \) is on the top of \( RS \) in \( s \) and \( pg \) is its parent subgoal then in the state \( \text{unfold}(s) \) all accessible subgoals \( g' \) such that \( pg \leq g' < g \) become deterministic, \( g \) is popped from \( RS \) and \( (!, 1, e) \) is put on \( AS \) \((e \) being the empty substitution). For a state \( s \) with \( g = (!, 1, e) \) on the top of \( AS \) the state \( \text{backtrack}(s) \) is obtained when \( g \) is popped from \( AS \) and \( (!, 1) \) is put on \( RS \).

The notion of equivalence of logical procedures to be defined below uses the following non-constructive operator.

Let \( lp/n \) be a logical procedure and \( pr = lp(\bar{W}) \) be some its instance. If \( \text{comp}(pr) \) is successful and \( \text{comp}(pr) = (s_0, ..., s_n) \) we set

\[
\text{image}(pr, 1) = \bar{W} \circ \text{con}(s_n)
\]

and proceed by \text{backtrack} on \( s_n \). So we obtain new computation \( \text{comp}(pr, 1) \). If it is again successful and \( \text{comp}(pr, 1) = (s_0, ..., s_n, ..., s_{n_1}) \), we set

\[
\text{image}(pr, 2) = \bar{W} \circ \text{con}(s_{n_1})
\]

and proceed by \text{backtrack} on \( s_{n_1} \) and so on either infinitely or up to the first \( i \) such that \( \text{comp}(pr, i) \) is infinite or unsuccessful. In these cases we set

\[
\text{image}(pr, j) = \omega
\]

for all \( j > i \).

**Definition 5** We say that two logical procedures \( lp_1/n \) and \( lp_2/n \) are equivalent \( (lp_1/n \equiv lp_2/n) \) i.f. for each \( n \)-tuple \( \bar{W} \) and for all \( j > 0 \)

\[
\text{image}(lp_1(\bar{W}), j) = \text{image}(lp_2(\bar{W}), j).
\]

**Definition 6** Let \( C_1 \) and \( C_2 \) be two classes of logical procedures. \( C_2 \) is a conservative extension of \( C_1 \) if \( C_1 \subset C_2 \) and for each \( lp_2 \in C_2 \) there is \( lp_1 \in C_1 \) such that \( lp_1 \equiv lp_2 \).

AS-machine is a formal base for definitions of time and space complexity to which we proceed now. Time is measured as the total number of steps of both types.

**Definition 7** Let \( lp/m \) be a logical procedure. Then its time complexity is the partial function

\[
time_{lp}(n) = \max\{li_{lp}(t_1, ..., t_m) + bt_{lp}(t_1, ..., t_m) \mid \sum_{i=1}^{m} |t_i| < n \},
\]

where \( li_{lp}(\bar{W}) \) and \( bt_{lp}(\bar{W}) \) are respectively the number of unfold and the number of backtrack steps in \( \text{comp}(lp(\bar{W})) \) if it is successful.
Space metrics on abstract stacks reflecting WAM optimizations are based on three simple notions:

**Definition 8** We call an accessible subgoal \( g \) in a state \( s \) a hypothesis if it has a descendant choice point. An accessible subgoal \( g \) in \( s \) is called founded if it has no resolvent sons to the right of the focus. An accessible subgoal which is not founded is called unfounded. A founded subgoal which is not a hypothesis is called proven.

Consider for example the state \( s \) in the figure 1. In this state the subgoal \((nlin(E^1, N^1), 2, u_2)\) is a choice point, hence a hypothesis. Moreover this subgoal is also unfounded because it has the son \((nlin(T^2, N^2), 0)\) in resolvent stack. On the other hand in the state in the figure 2 below all accessible subgoals except \((nlin_j(E^3, [T^3][S^3], N^3, R^3), 4, u_4)\) are proven. This example illustrates the effect of execution of the cut operator. It is only for accessible subgoal \((l, 1, e)\) that the subgoal \((nlin_j(E^2, [T^2][S^2], N^2, R^2), 3, u_3)\) becomes deterministic and (because it is founded) proven. As to \((nlin_j(E^3, [T^3][S^3], N^3, R^3), 4, u_4)\), it is not a hypothesis but it is unfounded.

\[
\begin{align*}
(nlin([p_1, p_2, p_3], N), 1, u_1) \\
(nlin_j(L^1, [], 0, N^1), 3, u_2) \\
(nlin_j(E^2, [T^2][S^2], N^2, R^2), 3, u_3) \\
(nlin_j(E^3, [T^3][S^3], N^3, R^3), 4, u_4) \\
N^1 is N^4 + 1 (nlin_j([], L^4, N^1, R^4), 0)
\end{align*}
\]

- \( u_1 \) is \( \{L^1 = [p_1, p_2, p_3], N^1 = N\} \),
- \( u_2 \) is \( \{L^1 = [E^2][T^2], S^2 = [], N^2 = 0, R^2 = N^1\} \),
- \( u_3 \) is \( \{E^2 = [E^3][T^3], S^3 = [T^2][S^2], N^3 = N^2, R^3 = R^2\} \),
- \( u_4 \) is \( \{L^4 = [T^3][S^3], N^4 = N^3, R^4 = R^3\} \).

**Fig. 2.** A state in the tail-recursive computation of \( nlin([p_1, p_2, p_3], N) \).

**Definition 9** Let \( pr = lp(t_1, ..., t_m) \) be a program and \( comp(pr) = (s_0, ..., s_n) \) be successful and \( s \) be a state in \( comp(pr) \). We define sizes of its stacks as follows:

\[
|AS(s)| = \sum_{g \in s} h(g), h(g) = \begin{cases} 1, & \text{if } g \text{ is a hypothesis} \\ 0, & \text{for all other subgoals} \end{cases}
\]
\[ |RS(s)| = \sum_{g \in gn} uf(g), \quad uf(g) = \begin{cases} 1, & \text{if } g \text{ is an unfounded subgoal}, \\ 0, & \text{for all other subgoals}. \end{cases} \]

\[ |US(s)| = \sum_{u \in US} w(u), \quad w(u) - \text{the weight of unifier } u. \]

\[ |UR(s)| = \max\{w(u) \mid u \in US\}. \]

Let \( u \) be the MGU used for elimination of a subgoal \( p(t_1, ..., t_k), j \) by a clause \( p(v_1, ..., v_k) : \lnot \text{body}. \) Then

\[ w(u) = \begin{cases} ts(u), & \text{if } (p(t_1, ..., t_k), j) \text{ is deterministic}, \\ ts(u) + fv(u), & \text{if it is a choice point}, \end{cases} \]

where \( ts(u) \) is the total size of all structures and lists unified with variables, \( fv(u) \) is the number of free variables in terms \( t_1, ..., t_k. \)

The workspace size of \( s \) \( ws(s) \) is defined as:

\[ ws(s) = |AS(s)| + |RS(s)| + |US(s)|. \]

ASM stack sizes induce partial space complexity functions of \( lp/m: \)

\[ as_{lp}(n) = \max\{|AS(s)| \mid s \in \text{comp}(lp(t_1, ..., t_m), \sum_{i=1}^{m} |t_i| < n\}, \]

\[ rs_{lp}(n) = \max\{|RS(s)| \mid s \in \text{comp}(lp(t_1, ..., t_m), \sum_{i=1}^{m} |t_i| < n\}, \]

\[ us_{lp}(n) = \max\{|US(s)| \mid s \in \text{comp}(lp(t_1, ..., t_m), \sum_{i=1}^{m} |t_i| < n\}. \]

Let us comment shortly these definitions. There is a simple relation between stacks of \( AS\)-machine and stacks of Warren Abstract Machine (i.e. local stack(s), global stack and trail). Stack of accessible subgoals is a model of that region of local stack which contains choice points and frozen activation frames. So \( as_{lp} \) can be viewed as a measure of nondeterminicity of \( lp\). Resolvent stack corresponds to the part of local stack containing activation frames of active subgoals. Unifiers stack size reflects the size of global stack (heap) and the size of trail. The measures reflect main \( WAM \) optimization rules. Namely, \( RS \) is not increased for founded subgoals, corresponding to last call optimization in \( WAM \); simultaneously it is taken into account that \( WAM \) does not create activation frames neither for unit clauses nor for clauses with one call in the body. Above this proven subgoals do not increase the size neither of \( AS \) nor of \( RS \). This reflects tail recursion optimization as well as local stack optimization while executing
cut operator. Thus \( r_{\text{st}} \) measures inherent recursion depth. Now we see that both \( n_{\text{lin}} \) and \( r_{\text{lin}} \) are bounded by constants for the definition in example 2 while they grow unbounded for the definition in example 1. Note that our estimate of \( US \) size is pessimistic because it reflects growth of global stack but does not reflect its reduction while garbage collection (only while backtracking). So, constant upper bound of \( US \) is absolute. But if \( US \) increases unlimited, some superficial for \( AS \)-machine semantics factors must be taken into account. For example, for programs not creating structures or lists unlimited growth of \( US \) reflects inevitable garbage collections and hence a delay. The predicate above \( n_{\text{lin}} J/4 \) is a typical example of such procedures. On the other hand procedures generating structures or lists require \( US \) space proportional to maximal depth of constructed terms. The treating of arguments modes and dependencies among variables like that of term-weight measures in [4] can substantially refine \(|US|\) and take into account garbage collection.

For any class \( C \) of Prolog programs we introduce complexity classes

\[
C(FA, FR, FU)
\]

of all programs in \( C \) with \( AS, RS \) and \( US \) bounded by functions in classes \( FA, FR \) and \( FU \) respectively. For theoretical purposes it is enough to consider only four features specific to Prolog as an algorithmic language: recursive definitions in form of Horn clauses, cut operator, terms over constants and lists (structures are superfluous) and finally, dynamic clauses controlled by built-ins \texttt{assert/1} and \texttt{retract/1}. Respectively, we consider four simplest subsets of standard Prolog with or without these means so that to observe their influence on time and space complexity. These are:

- kernel Prolog (\( \text{KP} \)), i.e. Prolog with lists and with the only built-in feature: the cut operator (!),
- kernel dynamic Prolog (\( \text{KDP} \)), i.e. kernel Prolog enriched by built-in predicates for controlling dynamic clauses,
- flat Prolog (\( \text{FP} \)) and
- flat dynamic Prolog (\( \text{FDP} \)), i.e. the subsets of kernel and respectively dynamic kernel Prolog without lists (i.e. with constants and variables as terms).

### 3 TIME COMPLEXITY

We explore time complexity of Prolog programs through time for solution existence problem, i.e. the problem of successful termination of a program. As it is clear this problem is unsolvable in \( \text{KP} \) and \( \text{KDP} \). It turns out to be unsolvable in \( \text{FDP} \) either.

**Theorem 1** [6] Solution existence problem is undecidable in \( \text{FDP} \) (even without cut).

The solution existence problem is certainly solvable in \( \text{FP} \). However it is exponentially hard.
Theorem 2 [6] The problem of validity of a closed formula in first order singular predicate logic is polynomially reducible to the solution existence problem in FP (without cut).

NEXPTIME is polynomially reducible to the validity problem in 1-SPL [7]. So we have an exponential nondeterministic time lower bound for the solution existence problem in FP.

Corollary 1 The solution existence problem in FP requires nondeterministic time $O(2^{c^N})$ for some $0 < \varepsilon < 1$.

On the other hand we find that a double exponential deterministic time upper bound is true for solution existence problem in FP.

4 SPACE COMPLEXITY

All logical procedures in FP and FDP are trivially space bounded. Let * and con denote the sets of all integer functions and constant integer functions, and i denote the singleton class containing the integer constant i ($i > 0$). We find the following fact in [6].

Proposition 1 $\text{FP}(*, *, *) = \text{FP}(1, 1, 1)$, $\text{FDP}(*, *, *) = \text{FDP}(1, 1, 1)$.

Of course such theoretical degeneracy of space measures in FP and FDP does not imply their inadequacy in these classes. In practice we are interested in complexity of a particular program and not of its optimal and maybe nonconstructive equivalent. The situation in KP and KDP is different.

Kernel Prolog is definitely that subset of Prolog in which space complexity should be explored since it contains main features specific to Prolog as a programming language. Programs in KP can exploit recursion of unlimited depth on lists which often creates problems with space. Typically in case of space deficiency the problem arises to find an equivalent tail recursive program, which is not always simple to do. So, the question naturally arises whether for each program in KP an equivalent "completely tail recursive" kernel Prolog program could be constructed. We give the positive answer to this question and simultaneously estimate space complexity of the solution. The following theorem proven together with my postgraduate student V.Chumakov shows that theoretically in KP only one choice point and a bounded number of resolvent subgoals are needed.

Theorem 3 KP is a conservative extension of KP(1, con, *).

This fact strengthens the theorem in [6] saying that 4 choice points are enough generally. Such a dramatic workspace optimization is reached through the use of a deterministic universal in KP logic procedure looking for a solution with a given "number" of a given program. This costs a square time delay of original computation. So, it cannot be regarded as practical. For practical purposes
program transformations $T$ are needed such that time complexity of programs $T(p)$ would be linear with respect to that of $p$ and $|AS|_{T(p)}$ and $|RS|_{T(p)}$ would be constant. In many special cases such “linear delay” transformations exist but we don’t know general linear delay transformation. Nevertheless the proof of theorem 3 shows that such a nondeterministic programming language as Prolog needs theoretically only one choice point for enumerating different program solutions through backtracking. From this follows an important corollary about functional programs (i.e. programs with no more than one solution) in KP.

**Corollary 2** For each functional program in KP there is a strongly equivalent deterministic program in KP.

Prolog programmers often use a fuzzy term “iterative program”. It means a program with control organized through backtrack loops. Mainly they mean repeat-loops which cost no space at all though sometimes choice point loops are meant too. So, it is not clear how could be this notion formalized in syntactic terms. However, we can define somewhat weaker notion in space complexity terms. The good idea could be to call iterative the programs with workspace bounded by a constant. This is however not rather reasonable in KDP because even separate unifications cost the size of unified structures which depends on query and logic procedure size. So, it is sensible to claim that iterative programs do not require space more then is needed for unifying arguments of calls.

**Definition 10** A program is iterative if its $|AS|$ and $|RS|$ functions are bounded by constants and $|US| = O(|UR|)$.

Dynamic control operators make possible general reduction of Prolog recursion to iteration in this sense. We find the following fact in [5, 6]:

**Theorem 4** For each deterministic program in KDP an equivalent iterative KDP program exists with unification rate $|UR|$ of the same order.

In fact the proof of this theorem gives a linear delay transformation of deterministic programs into iterative equivalents. Applying this transformation to the deterministic universal Prolog interpreter from our proof of theorem 3 we get a general recursion elimination theorem.

**Corollary 3** For each program in KDP there is a strongly equivalent iterative program in KDP.

5 APPLICATIONS OF ASM-COMPLEXITY

Facts above serve as theoretical background for complexity of Prolog programs. They illustrate correlations among different WAM workspace constituents. However ASM-complexity measures can be of real help for practical programming. Let us list some areas in which ASM-complexity measures can be helpful.
1. Being founded on simple and natural notions they are a good methodological base for teaching pragmatic styles of programming in Prolog.

2. ASM-complexity can be used for automatic static analysis of programming style and quality of programs. For example for determining for a given logical procedure whether it can overflow Prolog workspace on some data. One should not of course expect a complete solution of this problem because the problem of existence of a uniform constant upper bound is undecidable in KP for both $|AS|$ and $|RS|$. As far as we know the only one practically used sufficient condition of constant upper bound existence is "tail-recursivity". But it applies only to deterministic procedures. We can formulate much weaker criteria sufficient for existence of upper bounds on AS size and RS size. Moreover for "normal" procedures violation of these tests guarantee that the upper bounds do not exist.

3. ASM-complexity measures give multilateral view of space consumption and so are useful for estimating interrelations between various its aspects and computation time in practical situations. For example, the following program tries to guess satisfiable formulas from their skeletons of form $(Sub, Sub)$ or $not Sub$, where $Sub$ is either skeleton or a constant $p$:

**Example 3**

```prolog
?-sf(Skeleton, 1).
  sf(p, 0).
  sf(p, 1).
  sf([L, R], V) :-
    sf(L, LV),
    sf(R, RV),
    min(LV, RV, V).
  sf(not F, V) :-
    sf(F, V0),
    V is 1 - V0.
```

Its complexity analysis shows that in worst case $|AS| = 2^{2^{|RS|}}$ and $time_{sf}(N)$ is proportional to $2^{2^{|AS|}}$ and so to $2^N$ and $2^{2|RS|}$.

4. ASM-complexity measures can serve for estimating optimizing transformations. For example rather a straightforward method can be used for transforming linear programs to equivalent form with constant $|RS|$ at the cost of introduction of a list parameter representing current resolvent. For this transformation $T$ we have:

\[
AS_T(p) = O(AS_p), \quad time_T(p) = O(time_p) \quad \text{and} \quad RS_T(p) = O(1),
\]

but

\[
|US|_T(p) = O(|US|_p + |RS|_p * |UR|_p).
\]

However for many programs with "natural" recursion on lists or structures it can be done at the cost of linear expansion of $US_p$ by low factor constant as in the example 4. For the transformation in this example we have:
\[ AS_{tp} = O(AS_p), \quad US_{tp} = O(US_p) \text{ and } RS_{tp} = O(1), \]

though

\[ time_{tp}(N) = O(N \ast time_p(N)). \]

**Example 4**  
% a) A procedure on left-associative lists.

```prolog
p([E|T]) :-
  p(E),
  op(T).
p(E) :-
  op(E).
```

% b) Transformed procedure.

```prolog
tp([E|T]) :-
  deep_op([E|T],Rest),
  tp(Rest).
tp(E) :-
  op(E).
```

```prolog
deep_op([E|T],R) :-
  deep_op([E|T],Rest).
```

```prolog
deep_op([E,T],T) :-
  op(E).
```

Of course there may be various other applications of these measures because they reflect rather satisfactorily real resources needed by computer run Prolog programs.

**References**