On computational complexity of Prolog programs

A.Ja. Dikovsky

Institute for Applied Mathematics, Miusskaja Sq. 4, Moscow 125047, Russia

Abstract


Time and space complexity of Prolog programs is investigated in simplest Prolog subclasses: kernel Prolog (no structures, no built-ins), kernel dynamic Prolog (dynamic clauses control operators allowed), flat Prolog (the subset of kernel Prolog with no lists) and flat dynamic Prolog (the corresponding subset of kernel dynamic Prolog. Even in flat Prolog the problem of solution existence needs exponential time.

A new operational semantics of Prolog is introduced which allows measuring practical time and space consumption. The main space optimization result shows that any program in kernel Prolog can be transformed into an equivalent one with four choice points and the local stack bounded by a constant.

1. Introduction

A proper choice of adequate time and space complexity criteria for computations in Prolog needs thorough analysis. There is no problem in dealing with so complex problems that we do not distinguish between different polynomials. Any natural interpreter of Prolog can be viewed as a sort of multistack machine. So in this case we can think of time and space as abstract, just as for any other universal local-step automaton. However, for measuring complexity of real-practice Prolog programs we should look for specific and feasible criteria.

Time is usually measured in practice in the so-called “logical inferences” (LI). This measure is somewhat fuzzy. Sometimes LI corresponds to one unfold (or resolution) step, and sometimes to one unification of two terms. These are, as a matter of fact, of
the same order. However, for asymptotic time bounds we should normalize LI by
a factor of the rate of maximal size of terms in clauses unified in the resolution step.
So we should make distinctions between logical inferences and normalized logical
inferences (NLI). Distinguishing between unfold steps and backtrack steps could
become even more informative, thus measuring the whole time in LIs and BIs. In
any case
\[
\text{time}_{\text{BT}}(\text{computation}) \leq \text{time}_{\text{LI}}(\text{computation}).
\]
However, for deterministic programs \(\text{time}_{\text{BT}} = 0\), whereas for “nondeterministic” pro-
grams it is close to \(\text{time}_{\text{LI}}\).

Much more complex is the problem of the choice of a reasonable space complexity
measure for Prolog. The reason is that standard semantics founded on the SLD-
resolution rule [10] gives no terms to account for causes of space consumption in real
Prolog computations, because these causes are implementation-dependent. However,
after long evolution, especially after a Warren abstract machine (WAM) instructions
set was designed for compiling Prolog [15], there exists a de facto standard on Prolog
implementations. Almost all contemporary interpreters and compilers of Prolog use
standard stacks: a local stack(s) for activation frames and choice points, a trail for
backtrackable variables and a global stack (or heap) for lists and structures [15,13].
Besides these, most of them use standard recursion optimization rules: tail-recursion
optimization [2,14], last call or activation frame optimization [15], arguments
indexing [15,13], garbage collection [1], and so on. Real consumption of workspace
of a standard interpreter depends strongly on the recursion style. Very often, abso-
lutely logically correct and elegant Prolog programs run a computer out of space for
reasons expressible in extralogical and standard interpreter based terms. We illustrate
this thesis by very simple, but typical, examples. The first of them gives a definition
of screen representation of left-associative conjunctive normal-form propositional
formulae in the equivalent form without superficial brackets. For example,
\(((\langle e; e; e \rangle), (e; e)), (e; e))\) is transformed into \((e; e; e),(e; e),(e; e)\).

Example 1.1. \% wlf(+ Lcnf_formula).

\[
\begin{align*}
\text{wlf(C, D)} & : - \\
\text{wlf(C),} & \\
\text{write(’,’),} & \\
\text{wdj(D).} & \\
\text{wlf(D) : -} & \\
\text{wdj(D).} & \\
\text{wdj(D) : -} & \\
\text{write(’(‘),} & \\
\text{wd(D),} & \\
\text{write(’’)).} & 
\end{align*}
\]
On computational complexity of Prolog programs

wd((D1; D2)) :-
    wd(D1),
    write(';'),
    write(D2).
wd(D) :-
    write(D).

This definition seems to be the most natural because it directly follows the simplest context-free syntax rules for input formulae. However, it will overflow space for the local stack on a sufficiently long input formula, and it is impossible to explain this in logical terms. But instead of a pragmatically ineligible definition of wlf/1, we can use the following tail-recursive definition.

Example 1.2. % t_wlf(+ Lcnf_formula).
t_wlf((C, D)) :-
    w_deep_c((C, D), Rest),
    write(','),
    !,
    t_wlf(Rest).
t_wlf(D) :-
    t_wdj(D).
w_deep_c(((C1, C2), D), (Rest, D)) :-
    !,
    w_deep_c((C1, C2), Rest).
w_deep_c((D1, D2), D2) :-
    t_wdj(D1).
t_wdj(D) :-
    write(')'),
    t_wd(D),
    write(')').
t_wd((D1; D2)) :-
    w_deep_d((D1; D2), Rest),
    write(';'),
    !,
    t_wd(Rest).
t_wd(D) :-
    write(D).
w_deep_d(((D1; D2); D3), (Rest; D3)) :-
    !,
    w_deep_d((D1; D2), Rest).
w_deep_d((D1; D2), D2) :-
    write(D1).
The “tail” property of recursion demands that a recursive call should be last in the last alternative and that all subgoals from a parent recursive call to a child recursive call should become deterministic to the moment of the recursive call. As a matter of fact, the definition of $t_{\text{wlf}/1}$ is logically equivalent to that of $\text{wlf}/1$, but it is not dependent on the length of input formulae. Logical semantics does not explain this. Neither does it explain why (logically superfluous) cuts are needed in the definition of $t_{\text{wlf}/1}$, namely, why would Prolog run out of space again on long formulae without them. An optimization rule behind a standard interpreter transforms a tail recursion into an iteration excluding growth of local stack and trail. Examples 1.3 and 1.4 demonstrate the same effect for predicates not dealing with structures or lists. Example 1.3 gives the most naive definition of length of an input stream.

**Example 1.3.**

```
% strm_len(+ Input_stream_handle, - Stream_length).
strm_len(H, L) :-
    getO(H, -),
    strm_len(H, L0),
    L is L0 + 1.
strm_len(_,-,0).
```

Example 1.4 gives an equivalent tail-recursive definition.

**Example 1.4.**

```
% strm_t_len(+ Input_stream_handle, - Stream_length).
strm_t_len(H, L) :-
    strm_tl(H, 0, L).
strm_tl(H, Acc, L) :-
    getO(H, -),
    A is Acc + 1,
    strm_tl(H, A, L).
strm_tl(_, A, A).
```

Furthermore, $\text{strm}_{\text{len}}/2$ runs Prolog out of space for sufficiently long input streams, whereas $t_{\text{strm}}_{\text{len}}/2$ always succeeds. It is worth mentioning that such “infinite-loop” procedures can always be described equivalently by absolutely iterative backtrack-loops definitions. This is achieved at the cost of using global variables or facts (unit clauses) for parameters passing. Example 1.5 presents such a definition for the input stream length.

**Example 1.5.**

```
% strm_i_len(+ Input_stream_handle, - Stream_length).
strm_i_len(H, L) :-
    ctr_set(0,0), % built-in predicate setting
    repeat,        % global counter 0 to 0
```
readchar(H),
ctr_is(0, L). % built-in predicate unifying % L with counter 0 value
readchar(H) :-
get0(H, _),
ctr_inc(0, _), % built-in predicate incre-
! % menting counter 0 by 1
fail.
readchar(_).

This definition also succeeds for any input stream, again a fact that cannot be seen
from its purely logical interpretation. Although such a style may be regarded as
"awful" by those who write mostly metaprog languages (e.g. interpreters, compilers,
partial evaluators, etc.), it is widely used in application programming for implemen-
ting "infinite" loops, deeply embedded loops and for other purposes, and proves to be
rather efficient and helpful.

These examples demonstrate only three styles of writing Prolog programs: purely
logical, tail recursive and iterative and show that standard semantics of Prolog gives
no formal criteria of choice between them in concrete situations, although sometimes
performance or even fitness of programs depends on this choice.

Of course, it would be "contradictio in adjecto" if to write or estimate programs
in logic programming language, we should know details of its implementation.
Instead, we should use a very simple abstract model of a standard interpreter,
sufficient for adequately measuring workspace. We have presented such a model
called an abstract stack machine (ASM) in [6] and describe it here in detail. It is
founded on the notion of a derivation tree, representing a successful branch of an SL-D
tree. Derivation trees are represented naturally by three stacks: a stack of accessible
subgoals (AS), a stack of resolvent subgoals (RS) and a stack of unifiers (US). We
introduce space metrics on these stacks adequate for Warren abstract machine
implementations of Prolog. These metrics reflect, although not directly, the sizes of
the local stack, the trail and the global stack of WAM to within a constant factor. 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.

Another point to consider is the selection of subclasses of full Prolog for theoretical
analysis that is specific to Prolog space and time bounds. First, there is a de facto
standard subset of most Prolog dialects which evolved from Prolog DEC-10 [4], and
we should concentrate on this standard subset. Second, we should definitely abstract
from various means in the subset, external to Prolog itself but necessary in any
programming language, such as input/output or operation system interface built-ins.
Third, even this standard subset is too expressive in the presence of recursive
means and arithmetics. So we consider only four features specific to Prolog as an
algorithmic language: recursive definitions in the form of Horn clauses; cut operator;
terms over constants and lists (structures are superfluous); and the following dynamic clauses control built-ins:

assert/1, appending a new clause for a dynamic predicate p/n at the end of its definition, and
retract/1, deleting the first clause in the definition unified with the argument if there is one; if not, it fails.

Accordingly, we consider the four simplest subsets of standard Prolog with or without these means so as to observe their influence on time and space complexity. These are:

- *kernel Prolog (KP)*, i.e. Prolog with lists and with the cut operator (!) as the only built-in feature;
- *kernel dynamic Prolog (KDP)*, i.e. kernel Prolog enriched by built-in predicates for controlling dynamic clauses (assert/1, retract/1);
- *flat Prolog (FP)* and *flat dynamic Prolog (FDP)*, i.e. the subset of kernel and dynamic kernel Prolog, respectively, without lists (i.e. with constants and variables as terms).

This paper has four main sections. Section 2 contains the definitions forming abstract stack machine semantics of Prolog. In Section 3 we define and comment on Prolog complexity measures. In Section 4 we investigate the time complexity of a solution existence problem in Prolog. Here we find that even in flat Prolog (without cut) this problem needs $O(2^n)$ nondeterministic time, and in FDP it is unsolvable. It is well known (and seems to be folklore) that the solution existence problem is unsolvable in KP.

Section 5 is devoted to Prolog space complexity. In this revised version, several errors and inaccuracies unfortunately present in the preliminary publication [6] are rectified. In Section 5 we see that in FP and FDP we have trivially

$$FP(\ast, \ast, \ast) = FP(\text{con}, \text{con}, \text{con})$$

and

$$FDP(\ast, \ast, \ast) = FDP(\text{con}, \text{con}, \text{con}),$$

respectively ($\ast$ and con being the sets of all integer functions and constant integer functions, respectively). For kernel Prolog we show that KP is a conservative expansion of $\text{KP}(\text{con}, \text{con}, \ast)$, i.e. for any kernel Prolog program an equivalent kernel Prolog program can be constructed with stacks of accessible and resolvent subgoals bounded by constants.

We define *iterative* programs as the AS- and RS-bounded programs whose unification stack does not exceed the maximal unified term size. In [5] we have announced that for deterministic kernel dynamic Prolog programs recursion can be reduced to iteration in this sense. We outline the proof of this theorem in Section 5.
2. Abstract stack machine semantics

In this section we define operational semantics of Prolog programs through the notion of an abstract stack machine (AS machine). States of memory of the AS machine are exactly derivation trees.

Definition 2.1. A logical procedure is a set $lp/n$ of definitions of predicates with one distinguished predicate main/$n$. An instance of logical procedure $lp/t_1,\ldots,t_n$ with the set of definitions $lp/n$ and the query

\[ ?-\text{main}(t_1,\ldots,t_n). \]

Let $I_+$ denote the set of all positive integers and $I^*_+ = \{[i_1,\ldots,i_n]|i_j \in I_+, n \geq 0\}$ the set of all finite sequences of integers in $I_+$. Let $v_1 v_2$ denote the concatenation of sequences $v_1, v_2$ in $I^*_+$. We denote by $<$ the complete lexicographic order of sequences in $I^*_+$.

Definition 2.2. A finite subset $T \subset I^*_+$ is a tree if

(a) for any $v_1 v_2$ in $T$ $v_1$ is also in $T$, and

(b) for any $v$ and $j>1$ such that $v[j]$ is in $T$, $v[j-1]$ is also in $T$.

Sequences in $T$ are called nodes of $T$. The node $[]$ is a root of $T$. For a node $v$ the node $v[j]$ is called the $j$th son of $v$ and $v$ a parent of $v[j]$. A sequence of nodes $v_1,\ldots,v_n$, $n>0$, is a path (from $v_1$ to $v_n$) if for any $1 \leq i < n$ $v_{i+1}$ is a son of $v_i$. For a path $v_1,\ldots,v_n$ in $T$, $v_1$ is called an ancestor of $v_n$ and $v_n$ a descendant of $v_1$. A node is a leaf if it has no sons. For a node $v_1$ in $T$ any node $v$ in $T$ such that $v<v_1$ is a predecessor of $v_1$ (precedes $v_1$). Nodes $v$ such that $v<v_0$ (resp. $v>v_0$) are to the left (resp. to the right) of $v_0$.

Definition 2.3. A pair $t=(T,f)$, where $T$ is a tree and $f$ is a leaf of $T$, is called a focused tree; $f$ is called a focus 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 are the accessible ones, all others are resolvent. The subgoal $l(\[\])$ is called a query; variables in this subgoal (if any) are called query variables. We shall not differentiate in the sequel between nodes of states and the occurrences of subgoals corresponding to them.

Example 2.4. The focused labelled tree shown in fig. 1 is a state of the program $wlf((p1,(p2;p3)))$ resulting from the definition in Example 1.1 by substituting main/1 for $wlf/1$. 
Definition 2.5. Let \( s = \langle T, f, l \rangle \) be a state of a program \( lp(W) \) and \( (a_1, i_1, u_1), \ldots, (a_k, i_k, u_k), (a_{k+1}, i_{k+1}), \ldots, (a_n, i_n) \) be the sequence of all its subgoals in increasing order. Then the sequence \( (a_1, i_1), \ldots, (a_k, i_k) \) is called an accessible subgoals stack (AS), the sequence \( u_1, \ldots, u_k \) is called a unifiers stack (US) and the sequence \( (a_{k+1}, i_{k+1}), \ldots, (a_n, i_n) \) 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 \( \text{con}(s) = u_1 \circ \cdots \circ u_k \) of all substitutions in US is called a context of the state \( s \).

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 stacks. We define transitions from states to states in AS machines in terms of two operators on states: \textit{unfold} and \textit{backtrack}.

Definition 2.6. Let \( lp/m \) be a logical procedure, \( pr = lp(w_1, \ldots, w_m) \) be some of its instances and \( s \) be a state of \( pr \).

Operator \textit{unfold} applies to \( s \) if on the top of RS (i.e. in focus) there is a subgoal \( (p(V), d) \) and in the definition of \( p/n \) in \( lp/m \) there is an \( i \)th clause, \( i > d \), with the head \( p(o) \) unifiable with \( p(V) \circ \text{con}(s) \). Let \( u \) be the MGU for these two atoms and

\[
 r := r_1, \ldots, r_k, 
\]

\( k \geq 0 \), be an instance of the \( i \)th clause not containing variables in \( s \). Then the state \( \text{unfold}(s) \) results from \( s \) when \( (p(V), d) \) is replaced in RS by \( (r_1, 0), \ldots, (r_k, 0), (p(V), l) \) is put on top of AS and \( u \) is put on top of US (\( u \) eliminates \( p(V) \) by the \( i \)th clause for \( p/n \)).
Operator \textit{backtrack} applies to \( s \) when \textit{unfold} cannot be applied to it and if there is an accessible subgoal \((r, i)\) on the top of \( AS \), the functor in \( r \) is \( p/n \), the \( i \)th clause in the definition of \( p/n \) in \( lp/m \) is an instance of

\[ r := r_1, ..., r_k, \]

\( k \geq 0 \), and \( RS \) contains \((r_1, d_1), ..., (r_k, d_k), (q_1, d_{k+1}), ..., (q_l, d_l)\). In this case the state \textit{backtrack}(\( s \)) results from \( s \) when \((r, i)\) is popped from \( AS \), topmost \textit{MGU} \( u \) is popped from \( US \) and \( RS \) is transformed to \((r, i), (q_1, d_{k+1}), ..., (q_l, d_l)\).

The starting state of the \( AS \) machine on the program \( pr \) is the state \( s_0 \), in which \( AS \) and \( US \) are empty and \( RS \) contains \((\text{main}(w_1, ..., w_m), 0)\). The \textit{computation} of \( pr \) is the sequence \( \text{comp}(pr) = (s_0, s_1, ...) \) of states in which \( s_{i+1} = \text{unfold}(s_i) \) or \( s_{i+1} = \text{backtrack}(s_i) \) for each \( i \).

An accessible subgoal \((p(t_1, ..., t_n), d, u)\) in \( s_i \) with \( p/n \) defined in \( lp/m \) is called \textit{deterministic} if \( d \) equals the number of clauses in the definition of \( p/n \). Otherwise, such a subgoal is called a \textit{choice point}.

If \( \text{comp}(pr) = (s_0, ..., s_n) \), \( s_n = \text{unfold}(s_{n-1}) \) and \( RS \) is empty in \( s_n \) then \( \text{comp}(pr) \) is \textit{successful}. A \textit{result} of a successful computation of \( pr \) is the substitution \( \text{res}(pr) \), which is the restriction of \( \text{con}(s_n) \) to query variables (i.e. variables in \( w_1, ..., w_m \)). \( \text{comp}(pr) = (s_0, ..., s_n) \) is \textit{unsuccessful} if \( s_n = \text{backtrack}(s_{n-1}) \) and \( AS \) is empty in \( s_n \).

In order to define \( AS \)-machine semantics of a dialect of Prolog with a set of built-in predicates \( BIP \) we should define for each \( p/n \) in \( BIP \) \textit{unfold}(\( s \)) and \textit{backtrack}(\( s \)) on those states \( s \) where a subgoal \((p(o), d)\) is on the top of \( RS \) or a subgoal \((p(U), d, u)\) is on the top of \( AS \), respectively. For example, we define completely kernel Prolog if the following definition of cut operator \!/0 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 \( pg \preceq g' \preceq g \) become deterministic, \( g \) is popped from \( RS \) and \((!, 1, e)\) is put on \( AS \). For a state \( s \) with \( g = (!, 1, e) \) on the top of \( AS \) the state \textit{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 nonconstructive operator.

Let \( lp/n \) be a logical procedure and \( pr = lp(W) \) be some of its instances. If \( \text{comp}(pr) \) is successful and \( \text{comp}(pr) = (s_1, ..., s_n) \) we set

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

and proceed by \textit{backtrack} on \( s_n \). So we obtain a new computation \( \text{comp}(pr, 1) \). If it is again successful and \( \text{comp}(pr, 1) = (s_1, ..., s_n, ..., s_n) \), we set

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

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

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

for all \( j \geq i \).
Definition 2.7. We say that two logical procedures $lp_1/n$ and $lp_2/n$ are equivalent ($lp_1/n = lp_2/n$) if 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 2.8. Let $C_1$ and $C_2$ be two classes of logical procedures. $C_2$ is a conservative extension of $C_1$ if $C_1 \subseteq C_2$ and for each $lp_2$ in $C_2$ there is $lp_1$ in $C_1$ such that $lp_1 = lp_2$.

3. Complexity measures

As decided above, when counting time we should distinguish between logical inferences and backtrack steps.

Definition 3.1. Let $lp/m$ be a logical procedure, $pr = lp(\bar{W})$ be some of its instances and the computation $\text{comp}(pr) = (s_0, \ldots, s_n)$ be successful. Then $\text{li}_p(\bar{W})$ and $\text{bt}_p(\bar{W})$ are, respectively, the number of unfold and the number of backtrack steps in $\text{comp}(pr)$. $\text{time}_p(\bar{W}) = \text{li}_p(\bar{W}) + \text{bt}_p(\bar{W})$. Finally, for an integer $n > 0$
\[ \text{li}_p(n) = \min \left\{ \text{li}_p(t_1, \ldots, t_m) \mid \sum_{i=1}^{m} |t_i| < n \right\}, \]
\[ \text{bt}_p(n) = \min \left\{ \text{bt}_p(t_1, \ldots, t_m) \mid \sum_{i=1}^{m} |t_i| < n \right\}, \]
\[ \text{time}_p(n) = \min \left\{ \text{time}_p(t_1, \ldots, t_m) \mid \sum_{i=1}^{m} |t_i| < n \right\}. \]

The space metrics suggested here are founded on WAM instructions set [15], code copying and collecting garbage. Let us introduce the following classification of subgoals in a state of the AS machine.

Definition 3.2. 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 the state in Example 2.4. In this state the subgoal (wdj(D), 1, u3) is a choice point, and hence a hypothesis. Moreover, this subgoal is also unfounded because it has the son (write(‘’)), 0) in resolvent stack. On the other hand, in the state in Example 3.3 all accessible subgoals except (! \ wdj(D), 1, u2) are proven. This example illustrates the effect of execution of the cut operator. It is only for the accessible subgoal (!, 1, e)
that the subgoal \((t_{\text{wd}}(D), 1, u3)\) becomes deterministic and (because it is founded) proven. As for \((t_{\text{wdj}}(D), 1, u2)\), it is not a hypothesis but it is unfounded.

**Example 3.3.** The focused labelled tree shown in Fig. 2 is a state of the program \(t_{\text{wlf}}((p1; p2))\) resulting from the definition in Example 1.2 by substituting \(t_{\text{wlf}}/1\) for \(t_{\text{wlf}}/1\).

\[
\begin{array}{c}
(\text{write('('), 1, e) } \\
(t_{\text{wdj}}(D), 1, u2) \\
(\text{write(')'), 0) \\
(t_{\text{wd}}(D), 2, u3) \\
(w_{\text{deep}}(D1; D2), Rest), 2, u4) \\
(\text{write(';', 1, e) } \\
(!, 1, e) \\
(t_{\text{wd}}(Rest), 0) \\
(\text{write(D1), 1, u5})
\end{array}
\]

Fig. 2.

Here \(u1\) is the MGU \(\{D = (p1; p2)\}\), \(u2\) is \(\{D^1 = D\}\), \(u3\) is \(\{(D^1; D^2) = D^1\}\), \(u4\) is \(\{D^3 = D^1; D^2 = D^2 = \text{Rest}^2\}\), \(u5\) is \(\{D^4 = D^1^3\}\).

**Definition 3.4.** Let \(pr\) be a program, \(\text{comp}(pr) = (s_1, \ldots, s_n)\) be successful and \(s\) be a state in \(\text{comp}(pr)\). The **workspace** of \(s\), \(ws(s)\), is defined as

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

where

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

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

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

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

Let \(u\) be the MGU used for elimination of a subgoal \(p(t_1, \ldots, t_n)\) by a clause \(p(v_1, \ldots, v_n) \leftarrow \text{body.}\) Then

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

where \(ts(u)\) is the total size of all structures and lists unified with variables and \(fv(u)\) is the number of free variables in terms \(t_1, \ldots, t_n\).
We introduce five space complexity characteristics of a program \( pr \) with successful 
computation:

workspace size

\[
ws(pr) = \max \{ ws(s) \mid s \text{ in } \text{comp}(pr) \},
\]

accessible goals stack size

\[
as(pr) = \max \{ |AS(s)| \mid s \text{ in } \text{comp}(pr) \},
\]

corollary goals stack size

\[
rs(pr) = \max \{ |RS(s)| \mid s \text{ in } \text{comp}(pr) \},
\]

unification stack size

\[
us(pr) = \max \{ |US(s)| \mid s \text{ in } \text{comp}(pr) \},
\]

unification rate

\[
ur(pr) = \max \{ |UR(s)| \mid s \text{ in } \text{comp}(pr) \}.
\]

**Definition 3.5.** Let \( lp/m \) be a logical procedure. Five partial space complexity 
functions are connected to it:

\[
ws_{lp}(n) = \min \left\{ \text{ws}(pr) \mid \text{pr} = lp(t_1, \ldots, t_m), \sum_{i=1}^{m} |t_i| < n \right\},
\]

\[
as_{lp}(n) = \min \left\{ \text{as}(pr) \mid \text{pr} = lp(t_1, \ldots, t_m), \sum_{i=1}^{m} |t_i| < n \right\},
\]

\[
r s_{lp}(n) = \min \left\{ \text{rs}(pr) \mid \text{pr} = lp(t_1, \ldots, t_m), \sum_{i=1}^{m} |t_i| < n \right\},
\]

\[
us_{lp}(n) = \min \left\{ \text{us}(pr) \mid \text{pr} = lp(t_1, \ldots, t_m), \sum_{i=1}^{m} |t_i| < n \right\},
\]

\[
ur_{lp}(n) = \min \left\{ \text{ur}(pr) \mid \text{pr} = lp(t_1, \ldots, t_m), \sum_{i=1}^{m} |t_i| < n \right\}.
\]

For a class of logical procedures \( C \) and classes of integer functions \( A, R, U \) we 
denote by \( C(A, R, U) \) the class of those \( lp \) in \( C \) that there are functions \( a \) in \( A \), \( r \) in 
\( R \) and \( u \) in \( U \) such that

\[
as_{lp} \leq a, \quad rs_{lp} \leq r, \quad us_{lp} \leq u.
\]

We shall select two classes of functions; * – the class of all integer functions; 
\text{con} – the class of all constant integer functions. So, for example, \( KP(\text{con}, *, *) \) is 
the class of all logical procedures in \( KP \) with a stack of accessible subgoals bounded 
by constants.
Let us comment briefly on these definitions. There is a simple relation between stacks of the AS machine and stacks of the Warren abstract machine (i.e. local stack(s), global stack and trail). The stack of accessible subgoals is a model of that region of the local stack which contains choice points and frozen activation frames. The resolvent stack corresponds to the part of the local stack containing activation frames of active subgoals (i.e. subgoals included in the pointer chain starting from the topmost subgoal on the local stack (focus subgoal) and connected by pointers to the parent activation frame). The unifiers stack size reflects the size of the part of the global stack (heap) (component ts(u)) and the size of the trail. Definitions of complexity measures reflect main optimizations provided by WAM instructions. Namely, RS is not increased for founded subgoals, corresponding to last call optimization in WAM; simultaneously it is taken into account that WAM creates activation frames neither for unit clauses nor for clauses with one call in the body. Above this, proven subgoals increase the size neither of AS nor of RS. This reflects tail-recursion optimization as well as local stack optimization while executing cut operator. Note that our estimate of US size is pessimistic because it reflects growth of the global stack but does not reflect its reduction during garbage collection (only while backtracking). So the constant upper bound of US is absolute. But if US increases in an unlimited manner, some superficial AS-machine semantics factors must be taken into account. For example, for programs not creating structures or lists the unlimited growth of US reflects inevitable garbage collections and hence a delay. The predicate t_wlf/1 defined in Section 1 is a typical example of such procedures. On the other hand, procedures generating structures or lists require a US space proportional to the maximal depth of constructed terms. Here is an example of such a procedure.

Example 3.6. \% gen_list(+ Depth, - List).

\[
\begin{align*}
gen\_list(0,[ ]) & = \emptyset \\
gen\_list(D, [a | T]) & :- \\
  & D1 is D - 1, \\
  & gen\_list(D1, T).
\end{align*}
\]

AS-machine semantics reflects only those properties of the WAM instruction set which are expressible in terms of derivation trees. So it does not reflect, for example, static indexing of clauses heads. Besides this, proposed space consumption measures reflect real sizes of WAM stacks only to within a constant factor which may depend on an implementation. Nevertheless, this simple model is sufficient for practical Prolog programming. For example, space estimates explain the different behaviours of logically equivalent procedures in Examples 1.1–1.5. All the three stacks AS, RS and US grow proportionally to the length of input left-associative formulae for the procedure wlf/1. So it is no wonder that it overflows workspace for sufficiently deep input formula. On the other hand, as_wlf and rs_wlf are bounded by constants (cf. Example 3.3). Although us_wlf is not bounded, no terms in US are included in
a resulting term, so all of them are garbage. This shows that \textit{t_wlf} will run successfully on any formula fitting Prolog heap. However, this also means that real computation will be somewhat delayed by consecutive garbage collections. As shown by the space consumption analysis of stream length procedures in Examples 1.3–1.5, \textit{as_{strm_len}} and \textit{rs_{strm_len}} also grow proportionally with the length of the input stream. So it is not fit for long streams. And, again, \textit{as_{strm_i_len}} and \textit{rs_{strm_i_len}} are bounded by constants. As for \textit{us_{strm_i_len}} and \textit{us_{strm_i_len}}, our definition of \textit{US} size does not indicate their growth. Nevertheless, for some implementations they are growing too. In this case \textit{us_{strm_i_len}} must be preferred for long streams because its stacks sizes \textit{as_{strm_i_len}}, \textit{rs_{strm_i_len}} and \textit{us_{strm_i_len}} are bounded by constants.

4. Time complexity

We explore the time complexity of Prolog programs through time for the solution existence problem, i.e. the problem of successful termination of a program \textit{pr = lp(W)}. As is well known, this problem is unsolvable in \textbf{KP} and \textbf{KDP}. It turns out to be unsolvable in \textbf{FDP} too.

\textbf{Theorem 4.1.} \textit{Solution existence problem is undecidable in } \textbf{FDP}. \textit{Proof.} We show that for any \textit{N}-counter automaton \textit{A} there exists a Prolog program \textit{p_A} such that \textit{A} reaches its final state, being started on empty counters iff \textit{comp(p_A)} is successful. To this end, we describe a straightforward simulation of instructions of \textit{A} by clauses of \textit{p_A}.

We associate the predicate functor \textit{q_i/0} with each state \textit{q_i} of \textit{A} and the dynamic predicate functor \textit{c_i/0} with each counter \textit{i} and define the following correspondence between \textit{A} instructions and \textit{p_A} clauses:

(1) To an instruction

\[ q_1 \rightarrow q_2 + i \]

(transition from \textit{q_1} to \textit{q_2} and incrementing counter \textit{i} by 1) the clause

\[ q_1 :- \]

\[ \text{assert((c_{-i})}, q_2. \]

corresponds in \textit{p_A}.

(2) To an instruction

\[ q_1 \rightarrow q_2 - i \]

(transition from \textit{q_1} to \textit{q_2} and decrementing counter \textit{i} by 1) the clauses
On computational complexity of Prolog programs

q₁ :-
    retract(c_⁻⁻_i),
    !,
    q₂.
q₁ :-
    loop.

(3) To a conditional instruction

\[ i \rightarrow q₂ : q₃ \]

(transition from \( q₁ \) to \( q₂ \) if counter \( i \) is empty and to \( q₃ \) otherwise) the clauses

q₁ :-
    retract(c_⁻⁻_i),
    !,
    q₂.
q₁ :-
    q₃.

correspond. Besides these, there are two more clauses in \( p_A \):

loop :-
    loop.

and

qₜ.

for the finite state \( qₜ \) of \( A \).

We assume without loss of generality that

(i) \( A \) is deterministic (there are no different instructions with the same left parts, and states are partitioned into conditional (present in the left parts of conditional instructions) and unconditional ones),

(ii) if and when decrementing of a null counter is attempted, the computation of \( A \) fails.

So it is evident that the computation of procedure \( p_A \) with the query \(?-q₀,\ldots\) where \( q₀ \) is the starting state of \( A \), is successful if the computation of \( A \) with empty counters in \( q₀ \) reaches \( qₜ \). Moreover, we can omit cut operators in the clauses above if we assume without loss of generality that

(iii) for any state \( q \), save \( qₜ \), there is an instruction of \( A \) with \( q \) in its left part.

With this assumption no backtracking ever occurs.

As is well known, the termination problem for \( N \)-counter automata which we have now reduced to the solution existence problem in \( \text{FDP} \) is unsolvable for any \( N > 1 \) [12]. □
The solution existence problem is certainly solvable in $\text{FP}$. This follows from the following simple fact.

**Proposition 4.2.** For each program $pr$ in $\text{FP}$ there is a constant upper bound for the length of its possible successful computation.

**Proof.** Let $c$ be the number of constants in $pr$, $p$ the number of its predicate symbols, $n$ their maximal arity, $a$ the maximal number of alternatives in a predicate definition and $b$ the maximal number of calls in bodies of its clauses. Two atoms $g_1$ and $g_2$ are isomorphic if $g_1 = p(x_1, \ldots, x_m)$, $g_2 = p(\beta_1, \ldots, \beta_m)$, $x_i$ is a constant iff $\beta_i$ is the same constant, and variables $x_i$, $x_j$ are the same iff variables $\beta_i$, $\beta_j$ are the same. Any set of pairwise nonisomorphic atoms has not more than $p \Sigma_{i=1}^{p_n} C_i c^i 2^{a-i} \leq p c^{2n}$ elements. So in any state $s_i$ of a successful computation $\text{comp}(pr) = (s_1, \ldots, s_k)$ and in any path in $s_i$ from the root to a leaf there are not more than $ap c^{2n}$ subgoals. Therefore, $k$ cannot exceed $(ap c^{2n})^{poly}$. □

Although solvable, the solution existence problem in $\text{FP}$ is very complex because an exponentially hard problem can be polynomially reduced to it.

**Theorem 4.3.** The problem of validity of a closed formula in first-order singular predicate logic is polynomially reducible to the solution existence problem in $\text{FP}$.

**Proof.** We shall construct from any closed formula $\Phi$ of first-order singular predicate logic (i.e. logic with predicates of arity 1) a program $pr_\Phi$ in $\text{FP}$ such that $\Phi$ is valid iff $\text{comp}(pr_\Phi)$ is successful. We can assume without loss of generality that $\Phi$ has the prenex form

$$(Q_1 v_1) (Q_2 v_2) \cdots (Q_r v_r) F(\bar{V}),$$

where $\bar{V} = (v_1, \ldots, v_r)$ are all variables in $F(\bar{V})$, which in turn is a quantifierless formula in conjunctive normal form (any formula can be equivalently transformed in such a formula in polynomial time). Let $N$ be the number of predicate symbols in $F(\bar{V})$. Our construction uses partly the proof of the well-known fact that a formula $\Phi$ of 1-SPL with $N$ predicate symbols is valid iff it is valid in models of cardinality $2^N$ (see, for example, [3]). Let $p_1, \ldots, p_N$ be predicate symbols in $\Phi$ and $M$ be a model of $\Phi$ (denoted by $M \models \Phi$). Consider the following equivalence $\sim_\Phi$ on $M$: $m_1 \sim_\Phi m_2$ iff $\text{true}_{\text{value}}_M(p_1)(m_1) = \text{true}_{\text{value}}_M(p_2)(m_2)$ for all $1 \leq i \leq N$. We have $M \models \Phi$ iff $M/\sim_\Phi \models \Phi$, where $M/\sim_\Phi$ is the factor model of $M$ with respect to $\sim_\Phi$. $M/\sim_\Phi$ is isomorphic to the model $M_\Phi$ with elements of the form $(x_1, \ldots, x_N)$, $x_i$ in $\{0, 1\}$, $1 \leq i \leq N$, such that $\text{true}_{\text{value}}_{M_\Phi}(p_i)((x_1, \ldots, x_N)) = x_i$. So $M \models \Phi$ iff $M_\Phi \models \Phi$. We shall construct the program $pr_\Phi$ from $\Phi$ using its model $M_\Phi$. To this end, we relate to each variable $v$ in $F(\bar{V})$ the set of Prolog variables $M_1 v, \ldots, M_N v$, which we denote by $M_v$, and
assume that variables $M_{iv}$ are different for different $i$ and $v$. We introduce two constants 1 and 0 in $pr_{\Phi}$ and the following definitions:

$$b(1).$$

$$b(0).$$

$$\text{model}(X_1, \ldots, X_N) : -$$

$$b(X_1), \ldots, b(X_N).$$

$$\text{true}_{-p_i}(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_N). \quad \text{and}$$

$$\text{false}_{-p_i}(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_N). \quad \text{for all } 1 \leq i \leq N.$$

We proceed by induction on the structure of $\Phi$. Let

$$F(\bar{V}) = C_1(\bar{V}_1) \& C_2(\bar{V}_2) \& \ldots \& C_k(\bar{V}_k),$$

where each conjunct has the form

$$C_i(\bar{V}_i) = D_{i1}(v_{i1}) \lor \ldots \lor D_{il}(v_{il}),$$

$D_{ij}(v_{ij})$ being literals (positive or negative). Then

1. to each positive literal $D_{ij}(v_{ij}) = p_s(v)$ we relate the call $d_{ij}(M_v) = \text{true}_{-p_s}(\bar{M}_v)$,

2. to each negative literal $D_{ij}(v_{ij}) = \neg p_s(v)$ we relate the call $d_{ij}(M_v) = \text{false}_{-p_s}(\bar{M}_v)$,

3. to each conjunct $C_i(\bar{V}_i)$ we relate the definition

$$C_i(\bar{V}_i) : -$$

$$d_{i1}(M_{v_{i1}}).$$

$$\ldots$$

$$d_{il}(M_{v_{il}}).$$

4. to the formula $F(\bar{V})$ we relate the definition

$$f(M_{\bar{V}}) : -$$

$$c_1(M_{\bar{V}_1}), \ldots, c_k(M_{\bar{V}_k}).$$

The logical procedure constructed so far has the following property. Let $f(\bar{z})$ be the formula resulting from $F(\bar{V})$ by binding each variable $v$ in $F(\bar{V})$ by an object $x$ in $M_{\Phi}$ and, correspondingly, $f(\bar{z})$ from $f(M_{\bar{V}})$ by binding each variable in $M_v$ by the component of $x$ with the same number. Then $\text{truth-value}_{M_{\Phi}}(F)(\bar{z}) = 1$ iff the query

$$?f(\bar{z}).$$

is solvable. A contradiction of literals occurs in $F(\bar{z})$ iff unification corresponding to these literals variables fails in the computation of this query.

5. We relate to the subformula $(\exists v) \Psi(v, \bar{V})$ the definition

$$s(M_{\bar{V}}) :=$$

$$\text{model}(M_v),$$

$$g(M_v, M_{\bar{V}}),$$

$$\text{false}_{-p_i}(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_N). \quad \text{for all } 1 \leq i \leq N.$$
in which \( g(M_v, M_y) \) is the call constructed for the subformula \( \Psi(v, \bar{V}) \) and
(6) we relate to the subformula \( (\forall v) \Psi(v, \bar{V}) \) the definition

\[
\begin{align*}
&f_{\text{on}\_\text{all}}(M_y) :- \\
&f_{\text{on}\_\text{any}}(0, M_y).
\end{align*}
\]

\[
\begin{align*}
f_{\text{on}\_\text{any}}(1, M_y) :- \\
f(1, M_y).
\end{align*}
\]

\[
\begin{align*}
f_{\text{on}\_\text{any}}(M_v, M_y) :- \\
f(M_v, M_y), \\
succ_N(M_v, M_y), \\
f_{\text{on}\_\text{any}}(M_y, M_y).
\end{align*}
\]

where \( \bar{0} \) and \( \bar{1} \) are the \( N \)-tuples of 0 and 1, respectively, \( f(M_v, M_y) \) is the call constructed for the subformula \( \Psi(v, \bar{V}) \), and \( succ_N/N \) is the predicate defining for each \( N \)-tuple the next one and defined as follows:

\[
\begin{align*}
succ_{1}(1, 0), \\
succ_{m}(X_1, \ldots, X_{m-1}, 0, X_1, \ldots, X_{m-1}, 1). \\
&\quad \text{for all } N \geq m \geq 0, \\
succ_{m}(X_1, \ldots, X_{m-1}, 1, Y_1, \ldots, Y_{m-1}, 0) :- \\
&\quad succ_{m-1}(X_1, \ldots, X_{m-1}, Y_1, \ldots, Y_{m-1}). \\
&\quad \text{for all } N \geq m \geq 1.
\end{align*}
\]

So, after \( r \) steps we construct a variableless call \( h \) such that the query \( ?-h \). and the logical procedure constructed form the program \( pr_\Phi \) in question. It is readily seen that the size of \( pr_\Phi \) is polynomial with respect to the size of \( \Phi \). \( \square \)

As is known from [11], the problem of validity of closed formulae in 1-SPL requires nondeterministic time exceeding on infinitely many inputs \( S 2^{e|S|/\log |S|} \) for some \( e > 0 \). So we get the exponential nondeterministic time lower bound for the solution existence problem.

**Corollary 4.4.** The solution existence problem in FP requires nondeterministic time \( O(2^{cN}) \), for some \( 0 < c < 1 \).

As a matter of fact, \( 2^{cN/\log N} \) is the nondeterministic time upper bound for the validity problem in 1-SPL [9]. This shows that this problem is in \( \text{DTIME}(2^{2^c}) \). We find that a double exponential deterministic time upper bound is true for the solution existence problem in FP.

**Proposition 4.5.** The solution existence problem for a program \( pr \) in FP with \( c \) constants, \( p \) predicate symbols of maximal arity \( n \) and maximum \( b \) atoms in clauses can be solved by a deterministic procedure in time \( O(c^{kpkb^l}) \) for some \( k,l > 0 \).
**Proof.** This procedure implements an algorithm of interpretation of an input program, simultaneously constructing a two-dimensional table of lemmas LT. Lines of LT correspond to different subgoals \( g(W) \) and columns to different solutions of these subgoals. So there are at most \( O(pc^{2n}) \) lines and \( O(c^n) \) columns. Thus, the number of LT cells is at most \( O(pc^{3n}) \) and the size of a cell has a degree of \( O(n \log n) \).

**Algorithm.** When a subgoal \( g(W) \) is re-tried for the time \( k > 0 \),
1. it is tested not to be isomorphic to some of its ancestors and
2. then it is tested whether it already has the \( k \)th solution in LT:
   2.1) If there is a solution, it points only to it.
   2.2) If this solution is still absent in LT, \( g(W) \) is tried to being unfold just as it is done in the ASM, and in addition
      - if a new solution of \( g(W) \) is found (i.e. it is placed on top of AS) then this solution is included in LT,
      - if there are no alternatives for \( g(W) \), backtracking is effected.

*Time complexity:* (1) No more than \( O(|LT|) \) subgoals are unfolded.
(2) One unfold or backtrack step costs \( O(|LT|) \) time.
(3). Between two neighbour unfoldings, only the following are possible:
   - table solutions of postponed subgoals in RS,
   - backtracking through table-solved subgoals,
   - backtracking through unfolded deterministic subgoals.
Each subgoal can be visited no more than \( O(c^n) \) times. There are \( O(b|LT|) \) subgoals in AS and RS. So between two adjacent unfoldings at most \( O(|LT|^2c^{nb|LT|}) \) time can be spent. Thus, in total the time does not exceed \( O(|LT|^2c^{nb|LT|}) = O(c^{kpc^2n}) \) for some \( k, l > 0 \). Note that cut interpretation does not change this bound.  

**5. Space complexity**

All logical procedures in **FP** and **FDP** are trivially space-bounded since, for any program \( pr \) in one of these classes being an instance of logical procedure \( lp/n \) with \( c \) constants, not more than \( c^n! \) possible sequences of solutions exist. For each such sequence \( \sigma = (S_1, \ldots, S_k), k \geq 0 \), the program \( pr_\sigma \) can be defined with the query

\[
?-\text{main}(X_1, \ldots, X_n).
\]

and the logical procedure \( lp_\sigma/n \)

\[
\begin{align*}
\text{main}(S_1). \\
\vdots \\
\text{main}(S_k). \\
\text{main}(X) \leftarrow \\
\text{main}(\bar{X}).
\end{align*}
\]
Exactly one program \( pr_\sigma \) is equivalent to \( pr \) (although there could be no algorithm choosing the proper \( \sigma \) for \( pr \) as it follows, for example, from Theorem 4.1). As all three stacks of \( pr_\sigma \) are bounded by 1, we have the following proposition.

**Proposition 5.1.** \( \text{FP}(\ast, \ast, \ast) = \text{FP}(1, 1, 1) \), \( \text{FDP}(\ast, \ast, \ast) = \text{FDP}(1, 1, 1) \). (for an integer \( i \geq 0 \) \( i \) denotes the singleton class containing the constant \( i \)).

Of course, such theoretical degeneracy of space measures in \( \text{FP} \) and \( \text{FDP} \) does not imply their inadequacy in these classes. In practice, we are interested in the complexity of a particular program and not in its optimal and perhaps nonconstructive equivalent. Now we proceed to \( \text{KP} \) and \( \text{KDP} \), where the situation turns out to be somewhat different.

### 5.1. Space complexity of kernel Prolog

Kernel Prolog is definitely that subset of Prolog in which space complexity should be explored since it contains exactly the features specific to Prolog as a programming language. Programs in \( \text{KP} \) can exploit recursion of unlimited depth on lists, which often creates problems with space. Typically, in the case of space deficiency there arises the problem of finding an equivalent tail-recursive program, which is not always simple to do. So the question naturally arises whether, for each program in \( \text{KP} \), an equivalent “completely tail-recursive” kernel Prolog program could be constructed. We give a positive answer to this question and simultaneously estimate space complexity of the solution. The following theorem shows that in \( \text{KP} \) theoretically only four choice points and a bounded number of resolvent subgoals are needed.

**Theorem 5.2.** \( \text{KP} \) is a conservative extension of \( \text{KP}(4, \text{con}, \ast) \).

**Proof.** We must construct for each program \( pr \) in \( \text{KP} \) an equivalent program \( pr_0 \) in \( \text{KP} \) with the resolvent stack bounded by a constant and with not more than four choice points. First we introduce a coding of integers and programs in \( \text{KP} \) by lists. The coding of integers is evident: \([\ ] \) codes 0, and \( i \geq 0 \) is coded by the list of length \( i \) whose each element is \([\ ] \) (we denote it by \([i]\)). A logical procedure \( lp/1 \) is coded by the pair \( \{lp\} = [\text{index}(lp), \text{prog}(lp)] \), where both components are lists.

\[
\text{prog}(lp) = [\text{code}(\text{clause}_1), \ldots, \text{code}(\text{clause}_m)],
\]

where all the clauses of \( lp \) are listed in their natural order and clauses defining the same functor follow in succession. A fact \( \text{Fact} \) has the code \( \text{code}(\text{Fact}) = [\text{atom}(\text{Fact})] \), and a clause \( \text{Head} : \text{Body} \) has the code \( \text{code}(\text{Head} : \text{Body}) = [\text{atom}(\text{Head})] \text{atoms}(\text{Body}) \), \( \text{atom}(p(t_1, \ldots, t_n)) = [p[t_1, \ldots, t_n]] \), \( \text{atoms}((C_1, \ldots, C_r)) = [\text{atom}(C_1), \ldots, \text{atom}(C_r)] \) if \( r > 0 \). The index part of the program code is a table for the search of predicates definitions. \( \text{index}(lp) = [\text{func}(p_1/a_1), \ldots, \text{func}(p_n/a_n)] \), where \( p_i/a_i \) are all
predicates defined in \(lp\) and \(func(p_i/a_i)=[p_i,a_i,i,m_i]\), \(m_i\) being the number of alternatives in the \(p_i/a_i\) definition. So \(\{lp\}\) is a list. The counterpoint of this proof is the definition of a universal predicate solver/1 set forth in the appendix with the following property: for each program in \(KP\) \(pr=lp(W)\) and for each number \(i>0\)

\[
image(lp(W),i)=image(solver.\{lp\}(W),i),
\]

where \(solver.\{lp\}\) is the logical procedure with definitions:

\[
\begin{align*}
main(X_1,\ldots,X_n) :- & \quad \text{solution([\_],[SNum]), \% choice point} \\
& \quad \text{solver([\{lp\},SNum,[main,[X_1,\ldots,X_n]]]).} \\
\text{solution(N,Next)} & :- \quad \text{solution([\_],[N],Next).}
\end{align*}
\]

All predicates in the definition of solver/1 are tail-recursive. As the solver/1 predicate is applied to codes of \(KP\) programs we can express important metapredicates such as \(var/1\), \(nonvar/1\) and equality of unbound variables. To this end, we select two different special constants \(con_1\) and \(con_2\) (in the program in the appendix these are \(’$%&^1’\) and \(’$%&^2’\)) and eliminate their usage in program codes. This can easily be done by external to solver/1 means through a simple coding/decoding of program constants which we do not include here for the sake of simplicity. Having these constants we can introduce in the solver procedure the following definitions:

% unification/equality
% unify(?X,? Y)
unify(X, X).
% failure
fail :- unify(con1, con2).
% a constant
% mcon(?X).
mcon([\_]) :- !, fail.
mcon([\_\_]) :- !, fail.
mcon(X) :- unify(X, con1), !, fail.
mcon(_).
% a variable
% mvar(?X).
mvar(X) :-
not(not(unify(X, con1))). \% as fail/0 is present,
\text{eq vars(?X,? Y)} \% not/1 is available
eq_vars(X, Y) :-
   mvar(X),
   mvar(Y),
   not((unify(X, con_1),
        unify(Y, con_2))).

This enables us to define procedures of freezing and melting terms. Freezing is
a coding of terms by binding variables by special constants. It is used in all Prolog
interpreters. However, we need a tail-recursive definition of freezing, so we use
a somewhat unusual representation of frozen terms. They are represented through
a list of pairs of the form [Leaf_subterm, Path_to_this_leaf_from_the_root] ordered
lexicographically on paths. Such a “flat” representation of terms makes possible the
tail-recursive definitions of the predicates freeze_list/2 and melt_list/2
given in the appendix. (Both definitions were written by my student V. Chumakov.
They provide the proof of equivalence of solver.{lp} (W) to lpl(W), whereas the original
ones ensured the equivalence with respect to terms of bounded depth if not ground.)
So we define solver/1 as follows:

% solver([ + Program,
%          + Number,
%          + Query]).

solver([Program, Number, Query]) :-
   freeze_list(Query, Q),
   melt_list(Q, QQ),
   interp([Program, Number, Q, [QQ], [ ], [ ], QQ, Query]).

Here the interp/1 predicate implements a deterministic metainterpreter of a coded
procedure:

% interp([ + Program,
%          + Number,
%          + Query,
%          + Resolvent,
%          + Trace,
%          + Back_Trace,
%          + Cur_Query
%          - Result]).

The argument Resolvent serves as the resolvent stack and is initialized by
[Melted_query]. Current binding of query variables is available in the argument
On computational complexity of Prolog programs

Cur_Query. So when the solution counter Number equals 1 (i.e. [I]) the solution is obtained through the first alternative:

% getting solution
interp([Program,[I],Q,[I],Trace,[I],Query,Query]) :-
!

interp/1 uses two lists, Trace and Back_Trace, for emulation of accessible sub-goals stack. Trace is used on *unfold* steps implemented by the second alternative:

% unfold step
interp([Prog, I, Query, [Focus|Rest], Trace, [I], Q, Res]) :-
  move_focus([Prog, Focus, Trace, [[I]], [[I]]], New_Fc, New_Tr),
  interp([Prog, I, Query, New_Resolvent, New_Tr, [I], Q, Res]).

The predicate move_focus/1 is for trying in turn the clauses left, applying the chosen clause (if any) and moving the focus to a new position (see the appendix for its definition). This alternative is available when Back_Trace is empty. Most specific are the third to sixth alternatives used for *backtrack* steps emulation:

interp([Prog, I, Query, [Focus|Rest], Trace, [N|BT], Q, Res]) :-
  move_focus([Prog, Focus, Trace, N, New_Fc, New_Tr]),
  interp([Prog, I, Query, New_Resolvent, New_Tr, BT, Q, Res]).

% getting next solution
interp([Prog,[I]|I], Query, [I], Trace, [I], Q, Res) :-
  make_back_trace([Trace, Btrace]),
  melt_list(Query, QQ),
  interp([Prog, I, Query, [QQ], [I], Btrace, QQ, Res]).

% backtrack interpretation
interp([Prog, I, Query, _, Trace, _, Q, Res]) :-
  unify(Trace, [I]),
  unify(Btrace, [[[[I]], [I]], [[I]], [I]]),
  melt_list(Query, QQ),
  interp([Prog, I, Query, [QQ], [I], Btrace, QQ, Res]).
interp([Prog, I, Query, _, Trace, _, Q, Res]) :-
    make_back_trace([Trace, Btrace]),
    melt_list(Query, QQ),
    interp([Prog, I, Query, [QQ], [], Btrace, QQ, Res]).

There are two sources of failure: the unfold-failure and the next-solution-failure. When the unfold-failure occurs the fifth or sixth alternative fires. It melts the previously frozen copy of the initial query and either initializes (the fifth) or constructs from Trace (the sixth) the BackTrace list. This list is a mirror copy of the part of Trace from its beginning to the closest to the failure choice point. Deterministic subgoals are marked in it as [[*]], [Alternative_used]] and choice points are marked as [[[Alternative_used], [Alternative_used]]. BackTrace being not empty, the third alternative is taken until this list is exhausted. Through this alternative interp/l repeats its computation along Trace, thus reaching the proper choice point of backtracking. So the moment the point is reached it has an adequate copy of Trace and the query instance to the moment, while the alternative counter is incremented. As BackTrace becomes empty the unfold will be tried again through the second alternative, and the process repeats. The next-solution-failure occurs when the Resolvent becomes empty, whereas the solution counter Number is not yet equal to 1. In this case the fourth alternative is chosen which does just the same as the sixth and simultaneously decrements the solution counter. As one can easily see from the definitions in the appendix the solver/l is tail-recursive.

The constructed program pr0 finds first N solutions of pr through N backtracks choosing successive numbers l <= N and starting on them the solver/l procedure. If an incorrect solution number is tried the solver/l fails and pr0 loops.

It is clear that the resolvent stack is bounded because solver/l is tail-recursive and solution/2 does not increase this stack. So, to estimate the number of choice points at each moment we shall introduce the following measure on tail-recursive definitions.

Let p/n be a predicate with tail-recursive definition and a_max alternatives. We define for each alternative i a value g(p, i) and set

\[ g(p) = \max \{ g(p, i) \mid 1 \leq i \leq a_{\text{max}} \}. \]

I. Let p(U) :- guard p i, !, rest. be the ith alternative, i < a_{\text{max}}, guard_p_i being the maximal prefix of its body without cut operator call. We call it a guard prefix. We set g(p, i) = 0 if guard_p_i is empty and rest is either empty or is a recursive call p(V). Then we set g(p, a_{\text{max}}) = 0 if a_{\text{max}} is of the form p(U) :- p(V) or p(U). Further, we set

\[ bg(p, i) = \max \{ g(q) \mid q = p \land q \text{ is called in the ith alternative} \}. \]

Finally, we set inductively g(p, a_{\text{max}}) = bg(p, a_{\text{max}}) and for i < a_{\text{max}} g(p, i) = bg(p, i) + 1 if q(V) is called in guard_p_i for some q such that g(q) = bg(p, i); otherwise g(p, i) = bg(p, i). A predicate p and an alternative i for p is k-guarded if g(p) = k (resp. g(p, i) = k). g(p) is finite if p is ranked, i.e. there is a partial order \leq on predicates such
that if \( q_1 \) is a descendant of \( p \) and belongs to a guard prefix of his father \( q \) then \( q_1 \preceq q \). It is not difficult to prove that if all predicates of a program \( pr \) are not more than \( k \)-guarded then the computation of this program has at each moment not more than \( k \) choice points, and so \( as_{pr} \leq k \). The predicate \( \text{solver/1} \) is ranked and 3-guarded. As one more choice point is needed for solution/2, we have finally \( as_{pr_0} \leq 4 \).

This general recursion optimization theorem is somewhat related to the results in [7, 8], stating that for any kernel Prolog program there is an equivalent one with all predicates having the so-called "concluding recursion" definitions only, and that this concluding recursion is always reducible to an iteration scheme. The difference between our settings is that no space consumption estimate is established for the iteration scheme used there.

**Definition 5.3.** A logical procedure \( lp/n \) is **functional** if for any of its instances \( lp(\hat{W}) \) and for any \( i > 1 \) \( \text{image}(lp(\hat{W}), i) = \omega \).

The following corollary can be derived from Theorem 5.2.

**Corollary 5.4.** For any functional logical procedure \( lp_{1/n} \) in \( KP \) an equivalent 3-guarded logical procedure \( lp_{2/n} \) exists in \( KP(3, \text{con}, \ast) \).

### 5.2. Space complexity of kernel dynamic Prolog

Prolog programmers often use loosely the term **iterative program**. It means a program with control organized through backtrack loops. They actually mean repeat loops which cost no space at all, although sometimes choice point loops are meant too. So it is not clear how this notion could be formalized in syntactic terms. However, we can define a somewhat weaker notion in terms of space complexity. A good idea would be to call iterative the programs with workspace bounded by a constant. This is however rather unreasonable 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 more space than that needed for arguments unifications.

**Definition 5.5.** Let a program \( pr \) be an instance of a logical procedure \( lp/n \) in some classes \( C \). We call \( pr \) iterative if \( lp/n \) is in \( C(\text{con}, \text{con}, \ast) \) and for each solution of \( pr \) there is a constant \( c > 0 \) such that \( us_{pr} \leq c \ ur_{pr} \).

We shall show that recursion can be eliminated by iteration in this sense at least for deterministic \( KDP \) programs, i.e. those programs in whose computations choice points never arise and any backtrack step leads to computation failure. This means that right clauses are chosen in these computations only through unification of their heads against calls and not through backtracking. We have announced this theorem
earlier in [5]. All 0-guarded programs are deterministic. If the 0-guarded predicate solver/1 could be defined precisely, recursion could be eliminated in the whole KDP.

**Theorem 5.6.** For each deterministic program in KDP an equivalent iterative KDP program exists.

**Proof.** Let \( pr = lp(\bar{U}) \) be a deterministic program in KDP. We assume without loss of generality that a predicate name \( p_i \) specifies its arity \( a(p_i) \) uniquely and that the head of each static (i.e. not dynamic) clause contains all variables present in the clause. It is clear that having the dynamic clauses control predicates assert/1 and retract/1 we can express the predicate asserta/1:

\[
\text{asserta}((p(V) : Body)) \text{ appends the clause } p(V) : Body, \text{ at the beginning of the definition of the dynamic predicate } p/a(p).
\]

Let \( P \) be the set of all predicate names in \( pr \), \( ID \) be some set of identifiers, \( I \) be the set of positive integers and \( \langle \rangle \) be some injection from \( P \cup P \times I \times I \) into \( ID \) (so, \( \langle p \rangle \) and \( \langle p, i, j \rangle \) are identifiers in \( ID \)). We construct the equivalent to \( pr \) iterative program \( pr^* \) in two steps.

**Step 1:** We construct a logical procedure \( lp^p \) equivalent to \( lp \) in which the heads of some static clauses are replaced by propositional symbols (i.e. 0-arity predicates).

Let us first describe a transformation of clauses \( prop \) which we call a call propositionalization. Let alternative \( i \) for a predicate \( p \) have a call \( q(V) \):

\[
p(V) :-
\]
\[
\text{prefix},
\]
\[
q(U),
\]
\[
\text{suffix}.
\]

Then transformation \( prop \) replaces this clause by the clause

\[
p(V) :-
\]
\[
\text{prefix},
\]
\[
\text{asserta}(p^d(W)),
\]
\[
prop(p, i, q(\bar{U})),
\]
\[
\text{suffix}.
\]

where \( p^d \) is a new dynamic predicate of the same arity as \( p \) and \( prop(p, i, q(V)) \) is the sequence of calls

\[
\text{asserta}(q^d(V)),
\]
\[
\langle q \rangle,
\]
\[
\text{retract}(q^d(V)),
\]
\[
\text{retract}(p^d(W)),
\]
\[
\text{asserta}(p^d(W))
\]
On computational complexity of Prolog programs

$q(\bar{U}) :- \text{body}.$

the alternative $j$ is added in the definition of $\langle q \rangle$:

$$\langle q \rangle :-$$

$$q^a(\bar{U}),$$

$$\text{body}.$$ and the last alternative $\langle q \rangle :- \langle q \rangle.$ is introduced. The resulting logical procedure $lp'$ is equivalent to the original one. The equivalence of $lp$ and $lp'$ becomes clear if we think of new propositional symbols $\langle q \rangle$ as stacks for passing parameters. Then asserta($q^a(W)$) is interpreted as pushing the current state of $W$ on top of $\langle q \rangle$, whereas retract($q^a(W)$) is interpreted as popping a preceding state of $W$ from this stack. As $lp$ is deterministic, we are ensured that when the dynamic operators of prop($p, i, q(U)$) are called, the alternative $i$ is taken. So the global effect of these operators is adequate. In step 1 we transform all nonpropositional calls in static clauses of $lp$, thus obtaining an equivalent procedure $lp^\theta$.

Step 2: Let alternative $i$ in the definition of $\langle p \rangle$ in $lp^\theta$ be represented in the form

$$\langle p \rangle :-$$

$$p^a(W),$$

$$\text{pass\_parms\_p i 0}$$

$$p_1,$$

$$\text{pass\_parms\_p i 1}$$

$$\vdots$$

$$p_k,$$

$$\text{pass\_parms\_p i k}.$$ where $p_1, \ldots, p_k$ are all occurrences of propositional calls in its body and pass\_parms\_i\_j are the corresponding sequences of calls of dynamic operators between them. Then we introduce new propositional calls $\langle p, i, 0 \rangle, \ldots, \langle p, i, k \rangle$, a new dynamic fact stack/1 for resolvent goals and replace this clause by the new clause

$$\langle p \rangle :-$$

$$p^a(W),$$

asserta((stack(\langle p, i, k \rangle))),$$

$$\vdots$$

asserta((stack(\langle p, i, 0 \rangle))),

!, fail.
Then we introduce clauses defining new propositionals:

\[ \langle p, i, 0 \rangle : - \]
\[ p^q(W), \]
\[ pass\_parms\_p\_i\_0, \]
\[ asserta((stack(p_1))), \]
\[ !, \]
\[ fail. \]

\[ \langle p, i, j \rangle : - \]
\[ pass\_parms\_p\_i\_j, \]
\[ asserta((stack(p_{j+1}))), \]
\[ !, \]
\[ fail. \]

for all \( 1 \leq j < k \) and

\[ \langle p, i, k \rangle : - \]
\[ pass\_parms\_p\_i\_k, \]
\[ !, \]
\[ fail. \]

The query \(?-q(\bar{U}).\) is transformed into \(?-main(\bar{U}).\) and new clauses defining main/a(main) are added:

\[ main(\bar{U}) : - \]
\[ asserta((q(\bar{U}))), \]
\[ asserta((stack([\text{bottom}]))), \]
\[ asserta((stack(<q>))), \]
\[ inf\_loop, \]
\[ retract(q(\bar{U})). \]

\[ inf\_loop : - \]
\[ repeat, \]
\[ step. \]

\[ step : - \]
\[ retract(stack([\text{bottom}])), \]
\[ !. \]

\[ step : - \]
\[ retract(stack(G)), \]
\[ G. \]

The constructed program \( pr^* \) is equivalent to \( pr. \) It effects the same unifications as \( pr \) and in the same order. One unification is effected at one repeat-loop step. Parameters are passed through dynamic facts accessed in the first-in–first-out mode.
Moreover, the resolvent is not represented as a list but as a dynamic predicate stack/1 whose clauses contain resolvent goals and are accessed also as a stack (first fact on its top). Being a repeat-loop pr* takes AS-space and RS-space bounded by a constant. Moreover, the whole US-space needed for \( \text{comp}(pr*) \) is bounded by maximal US-space needed for one loop step. So, \( \text{us}_{pr*} \leq c \text{ us}_{pr} \) for some constant \( c > 0 \).

6. Conclusion

Computation time bounds show that even in minimal Prolog subsets the solution existence problem cannot be solved better than by a trivial brute search algorithm.

Space optimization results in this paper show that, in general, both recursion control stacks can be bounded by small integers with the help of minimal and standard Prolog means. Of course such “space optimization” must be regarded only as a theoretical background because it is achieved at the cost of high delay. Nevertheless, we are convinced that the AS-machine semantics and introduced space consumption measures are adequate for estimating Prolog programs in practice.

Acknowledgment

We gratefully acknowledge Mats Carlsson from SICS, Paul Tarau from Université de Moncton and François Bry from ECRC for stimulating discussions on our Prolog workspace metrics and for their criticism of the illustrations therein.

Appendix

A.1. Universal tail-recursive interpreter

```prolog
% Universal in KP predicate
% solver([ +Program, % coded program
% +Number, % solution number
% +Query ]). % query
solver([Program, Number, Query]) :-
  freeze_list(Query, Q), % freezing query
  melt_list(Q, QQ), % initializing resolvent
  interp([Program, Number, Q, QQ, Query]). % by melted query

% Tail-recursive metainterpreter
% interp([ +Program, % [Index,Prog_Code]
% +Number, % solution number
% +Query, % frozen query
% +Resolvent, % current resolvent
% +Trace, % interpreter trace
```
% Back_Trace,      % backtrack trace 
% Chg_Query        % query accumulator
% Result          % resolved query

% getting solution
interp([Program, [I], Q, [ ], Trace, [ ], Query, Query]) :-
  !.

% unfold step
interp([Prog, I, Query, [Focus|Rest], Trace, [ ], Q, Res]) :-
  move_focus([Prog, Focus, Trace, [[I]], [], NewFc, NewTr]),
  % [[I]], [[ ]] 1-st alternative "number"
  concat(NewFc, Rest, NewResolvent),
  !,
  interp([Prog, I, Query, New.Resolvent, New.Tr, [ ], Q, Res]).
interp([Prog, I, Query, [Focus|Rest], Trace, [N|BT], Q, Res]) :-
  move_focus([Prog, Focus, Trace, N, NewFc, NewTr]),
  concat(NewFc, Rest, NewResolvent),
  !,
  interp([Prog, I, Query, New.Resolvent, New.Tr, BT, Q, Res]).

% getting next solution
interp([Prog, [I], Query, [ ], Trace, [ ], Q, Res]) :-
  make_back_trace([Trace, Btrace]),
  melt_list(Query, QQ),
  !,
  interp([Prog, I, Query, [QQ], [ ], Btrace, QQ, Res]).

% backtrack interpretation
interp([Prog, I, Query, _, Trace, _Q, Res]) :-
  unify(Trace, [I]),
  unify(Btrace, [[[I]], [], [[I]], [[I]]]),
  melt_list(Query, QQ),
  !,
  interp([Prog, I, Query, [QQ], [ ], Btrace, QQ, Res]).
interp([Prog, I, Query, _, Trace, _Q, Res]) :-
  make_back_trace([Trace, Btrace]),
  melt_list(Query, QQ),
  interp([Prog, I, Query, [QQ], [ ], Btrace, QQ, Res]).

% interpretation step
% move_focus([ +[Index,Prog],
%   [+Focus_subgoal,
%    +Trace,
%    +Number_of_alternative,
%    New_Focus,
%    -New_Trace]).
% + marks the end of clause body in resolvent and trace,
% * marks deterministic subgoals in trace.
% + is transferred from resolvent to trace where it marks
% the border of proof subtree for an accessible subgoal.
move_focus([[-,-], +, Trace, _[ ], [[+]|Trace]]) :-
  !.

% cut interpretation
move_focus([[...],[!'!', Trace,...],[[[*],[[]]],N.trace]]) :-
  cut_goals([[[]], Trace,N.trace]).

% unfolding a subgoal
move_focus([[Index, Prog],
              [Fn|Args], % subgoal in focus
              Trace,
              [Dn, N]], % Dn = * if the subgoal is deterministic, else Dn = N (current alternative number)
              Mbody,
              Ntrace]) :-
  len(Args, Arity),
  memb([Fn, Arity, Loc, An], Index),
  try_clause(An, Dn, N, Loc, Prog, [Fn|Args], Trace, Ntrace, Mbody).

% choice of a clause to apply
% try_clause(+Number_of_alternatives,
%              +Determinacy_tag,
%              +Current_alternative_number,
%              +Definition_location,
%              +Prog,
%              +Subgoal,
%              +Trace,
%              -New_Trace,
%              -Marked_body).
try_clause(An,..., An, Loc, Prog, Goal, Trace, Ntrace, Mbody) :-
  % last alternative
  apply([*], An, Loc, Prog, Goal, Trace, Ntrace, Mbody), !.
  try_clause(An,..., Num, Loc, Prog, Goal, Trace, Ntrace, Mbody) :-
  % deterministic subgoal
  apply([*], Num, Loc, Prog, Goal, Trace, Ntrace, Mbody), !.
  try_clause(An,..., Num, Loc, Prog, Goal, Trace, Ntrace, Mbody) :-
  % trying current
  apply(Num, Num, Loc, Prog, Goal, Trace, Ntrace, Mbody), !.
  try_clause(An,..., Num, Loc, Prog, Goal, Trc, Ntcr, Mbd) :-
  % incrementing alternative counter
  try_clause(An,..., [Num], Loc, Prog, Goal, Trc, Ntcr, Mbd).

% goal elimination
% apply(+Determinacy_Tag,
%        +Curr_Alternative_Num,
%        +Def_Location,
%        +Prog_Code,
%        +Goal_in_Focus,
%        +Trace,
apply(Dt, Num, Lot, Prog, Goal, Trace, Ntrace, Mbody) :-
  clau(Num, Lot, Prog, [Goal]),
  unify([Marked_Body]),
  unify([-], [Dt, Num|[Trace], Ntrace]).

apply(Dt, Num, Lot, Prog, Goal, Trace, Ntrace, Mbody) :-
  clau(Num, Lot, Prog, [Goal|Body]),
  unify([-], [Dt, Num|[Trace], Ntrace]).

% clause extraction
% clau(+Clause_Numb,+Def_Location,+Prog,-Clause_Pattern).
clau(Num, Lot, Prog, ClauPattern) :-
  concat(Num, Lot, [], IPosition),
  nth_memb(Position, Prog, Clause),
  melt_list(Clause, ClauPattern).

% make_back_trace(+Trace, % trace for forward chaining
% -Btrace)). % trace for backtrack emulation
% Cuts the trace up to closest choice point and increments its
% alternative counter by 1
make_back_trace([Trace, Btrace]) :-
  !, fail.
make_back_trace([[+|[Trace], Btrace]]) :-
  !, make_back_trace([Trace, Btrace]).
make_back_trace([[|[Trace], Btrace]]) :-
  !, make_back_trace([Trace, Btrace]).
make_back_trace([[+[|][Trace], Btrace]]) :-
  !, make_back_trace([Trace, Btrace]).
make_back_trace([[[],[Trace], Btrace]]) :-
  !, make_back_trace([Trace, Btrace]).
make_back_trace([[L, EI][Trace], Btrace]) :-
  mk_trace1([[L, [EI]I][Trace], [L, Btrace]].

% mk_trace1(+Trace,+Acc,-Btrace1).
% eliminates marking of leaves and bodies behind choice point
mk_trace1([], L, L)) :-
  !.
mk_trace1([|], L, Res)) :-
  !, mk_trace1([Trace, L, Res]).
mk_trace1([],[|], L, Res)) :-
  !, mk_trace1([Trace, L, Res]).
mk_trace1([[EI], E2][Trace], L, Res)) :-
  mk_trace1([[EI, E2][L], Res]).
% making subgoals deterministic in the scope of cut
% cut_goals([ +Depth,+Trace,-New_Trace ]).  
% Depth is the depth of a subgoal with respect to the father  
% of cut operator. Its values:
%   [ ] - out of the scope  
%   [[ ]],[[ ],[ ]] should be made deterministic  
%   greater then [[ ],[ ]] should not.

cut_goals([ [ ], Trace, Trace ]) :-  
  !.
cut_goals([[ ]], [[-], [S, L]|Trace], [[-], [[+], L]|Ntrace]) :-  
  % brother-leaves
  !,         
  cut_goals([[[ ]], Trace, Ntrace]).
cut_goals([Depth, [[-], L|Trace], [[-], L|Ntrace]]) :-  
  % other leaves
  !,         
  cut_goals([Depth, Trace, Ntrace]).
cut_goals([Depth, [[+]|Trace], [[+]|Ntrace]]) :-  
  % body end; depth increases
  !,         
  cut_goals([[[ ]]|Depth], Trace, Ntrace)).
cut_goals([[[ ]], [[S, L]|Trace], [[[+], L]|Trace]]) :-  
  % the father of cut
  !.
cut_goals([[[ ], [ ]], [[S, L]|Trace], [[[+], L]|Ntrace]]) :-  
  % not leaf brother
  !,         
  cut_goals([[[ ]], Trace, Ntrace]).
cut_goals([[[ ]]|Depth], [L|Trace], [L|Ntrace]) :-  
  % out of scope of cut
  cut_goals([Depth, Trace, Ntrace]).

% terms freezer
% freeze_list(+Term,-Frozen_term).
freeze_list(L, S) :-  
  freeze('$$%&'~2', L, L1, [ ], [ ], [ ]),  
  revcomp(L1, S).

% freeze(+Head, % current subtree  
% +Stack, % stack for postponed subterms  
% -Result, % resulting frozen list  
% +List, % accumulator for the result  
% +Path, % path to current subtree  
% +Var ). % accumulator for variable codes
% constants: '"%&'~2' marks the end of a term; as the first  
% argument causes taking a term from stack  
% '"%&'~3' marks tail of a list  
% a variable
freeze(H, T, S, R, C, V) :-  
  avar(H),
% a list variable
freeze('%%&`2', T, S, [[K, [C]]][R], R, [C], V) :-
  avar(T),
  code_var(T, K, [[[ ]][C]], V, NV),
!.

% a variable-head of a list
freeze('%%&`3', [H][T], S, R, [|[1][C1][C2], V) :-
  avar(H),
  code_var(H, K, [[[ ]][C1][C2], V, NV),
  freeze('%%&`2', T, S, [[K, [C][C2]]][R], [[[ ]][C1][C2], NV).
%
% a list-head of a list
freeze('%%&`3', [[H][T]][I], S, R, [C, V) :-
  empty(T1), % here and below ; is used
  unify(T, TL); % only for economy of text
  unify('%%&`3', T1[T], TL)),
%
% end of a list
freeze('%%&`2', [H][T], S, R, [C], V) :-
  acon(H),
  unify(H, '%%&`2'),
!.
%
% ascending sublist path
freeze('%%&`2', [H][T], S, R, C, V) :
  unify(C, [D][L]);
  unify(C, D),
  unify(L, [ ]));
  unify(Cn, [[[ ]][D]][L]),
!.
freeze(H, T, S, R, Cn, V).
%
% the end
freeze('%%&`2', [ ], S, S, _) :-
!.
%
% a constant or [ ]
freeze(H, T, S, R, C, V) :-
  acon(H),
  unify(H, [ ])),
!.
freeze('%%&`2', T, S, [[H, C][R], C, V).
%
% descending sublist path
freeze([[H][L], T, S, R, C, V) :-
  empty(L),
  unify('%%&`2'[T], TL),
  concat('%%&`3', L, '%%&`2'[T], TL)),
!.
freeze(H, TL, S, R, [[ ]][C], V).
% freezing a variable
\% code_var(-Var, % a variable
% -Key, % its code (the path to first occurrence
% +Path, % its path
% +List, % variable codes list
% -New_List).
% first occurrence
code_var(H, C, [ ], [[H, C]]) :-
!.
% one more occurrence
code_var(H, K K, C, [[H H, K K]|L], [[H H, K K]|L]) :-
\quad eq_vars(H, HH),
\quad !.
code_var(H, K, C, E[R], [E|T]) :-
\quad code_var(H, K, C, R, T).

% melting a list
% melt_list(+Frozen_List,-Melted_list).
melt_list(F L, List) :-
  memb([_, [E..]], F L),
  succ(E, Ei),
  melt_flat(F L, [ ], [ ], Ei, List).

% melt_flat(+Frozen_List, % tail of flat frozen list
% +Variables_List, % accumulator for met variables
% +List, % accumulator for the result
% +Last_Path, % last melted subterm path
% -Result).
% the end
melt_flat([ ], _, Res, _, Res) :-
!.
% accumulator initialization
melt_flat([[C, Path]|T], V L, [ ], Past, Res) :-
  !,
  width_step([C, Path], V L, [ ], Past, New, N vl),
  melt_flat(T, N vl, New, Path, Res).
% subterm inclusion
melt_flat([[C, Path]|T], V L, List, Last, Res) :-
  width_step([C, Path], V L, List, Last, New_list, N vl),
  melt_flat(T, N vl, New_list, Path, Res).

% search for path to include a subterm
% width_step(+[Code,Path], % subterm code and path
% +Var_List, % accumulated variables
% +List, % constructed list
% +Former, % path of last inclusion
% -Result,
% -New_Var_List),
% path found
A.Ja. Dikouskq

width_step([C, [P]], VL, List, [New|List], N_vl) :-
  !,
  depth_step([C, [P]], VL, New, N_vl).
% search continued
width_step([C, [F|Path]], VL, [First|L], [F|Form], [New|L], N_vl) :-
  !,
  width_step([C, Path], VL, First, Form, New, N_vl).
% subterm inclusion
width_step([C, [F|Path]], VL, List, Past, [[New|List], N_vl) :-
  depth_step([C, Path], VL, New, N_vl).

% inclusion of a subterm
% depth_step(+[Code, Path], % subterm code and path
%  +Var_List, % accumulated variables
%  -Term, % melting primitive subterms
%  -New_Var_List).
% melting primitive subterms
depth_step([C, []], VL, Term, N_vl) :-
  !,
  melt_leaf(C, VL, Term, N_vl).
% sublist construction
depth_step([C, [Path]], VL, [Term], N_vl) :-
  depth_step([C, Path], VL, Term, N_vl).

% melting primitive subterms
% melt_leaf(+Code, % primitive code
%  +Var_List, % accumulated variables
%  -Result, % empty list or a constant
%  -NewVarList).
% old variable
melt_leaf(Code, VL, Var, VL) :-
  memb([Var, Code], VL),
  !.
% new variable
melt_leaf(Code, VL, Var, [Var, Code]|VL).

A.2. Definition of program coding
% Coder of Prolog clauses into list form
% term_coder(+Term,-List,+Switch(*/+))
% Switch values: * fact/subgoal A; coded into [[A]]
% + data-term
term_coder(Term, [LH|LR], *) :-
  Term = ..([|), Head, Rest],
On computational complexity of Prolog programs

(atomic(Head),
  [Head] = LH;
  term_coder(Head, LH, +)),
!,
term_coder(Rest, LR, +).
term_coder(O, [O], +) :-
  (var(O);
   atomic(O)),
!,
term_coder(O, O, +) :-
  (var(O);
   atomic(O)),
!,
term_coder([Head|Rest], [LH|LR], S) :-
  term_coder(Head, LH, +),
!,
term_coder(Rest, LR, +).
term_coder(Term, [IH|JR], S) :-
  Term = ..[\',\',Head,Rest],
  (atomic(Head),
   [Head] = LH;
   term_coder(Head, LH, +)),
!,
term_coder(Rest, LR, S).
term_coder(Term, [LH, LR], +) :-
  Term = ..[LH|Rest],
  atomic(LH),
!,
term_coder(Rest, LR, +).
term_coder(Term, [[LH, LR]], +) :-
  Term = ..[LH|Rest],
  atomic(LH),
term_coder(Rest, LR, +).

% index construction
% make_index(+Old_Index,+Element,-New_Index,+Position)
make_index(Acc, [[F|Args]..], Nacc, C) :-
  !,
  len(Args, Arity),
  ins_index(F, Arity, C, Acc, Nacc).
make_index(Acc, [F..], Nacc, C) :-
  ins_index(F, [ ], C, Acc, Nacc).

% inserting elements in index
% ins_index(+F,+A,+C,+Old_Index,-New_Index)
% F/A a functor; C location of F/A in program code.
% index element is [F,A,C,N], N - the number of alternatives
% in the definition of F/A.
ins_index(F, A, C, [ ], [[F, A, C, [ ]]]) :-
  !.
ins_index(F, A, C, [[F, A, Old, N]|List], [[F, A, Old, [ ]|N]]|List) :- !.
ins_index(F, A, C, [First|Old], [First|New]) :-
     ins_index(F, A, C, Old, New).

A.3. Library predicates

% unification/equality
% unify(?X,?Y)
unify(X, X).
% failure
fail:-
     unify('$$\&^\_1', '$$\&^\_2').

% list emptiness
empty(X) :-
     avar(X), !, fail.
empty(X) :-
     unify(X, [ ]).

% a constant
% acon(?X).
acon([ ]) :- !, fail.
acon([.-.]) :- !, fail.
acon(X) :-
     unify(X, '}$%&^\_1'), !, fail.
acon(_).

% a variable
% avar(?X).
avar(X) :-
     not(not(unify(X, '}$%&^\_1'))).

% same variables
% eq_vars(?X,?Y)
eq_vars(X, Y) :-
     avar(X), avar(Y),
     not((unify(X, '}$%&^\_1'), unify(Y, '}$%&^\_2')).
On computational complexity of Prolog programs

% lists as integers
% succ(+X, -(X+1)).
% prec(+X, -(X-1)).
succ(D, [[ ]|D]).
pred([[ ]|D], D).
% membership check
memb(X, [X|..]) :-
    !.
memb(X, [..|Y]) :-
    memb(X, Y).

% N-th member of a list
% nth_memb(+N, *List, -Elem)
nth_memb([[[ ]]|], [Elem|..], Elem) :-
    !.
nth_memb([[ |N]|], [..|Tail], Elem) :-
    nth_memb(N, Tail, Elem).

% lists concatenation
concat([], Y, Y) :-
    !.
concat([X|W], Y, [X|Z]) :-
    concat(W, Y, Z).

% length of a list
% len(+List, -Length)
len([], [ ]) :-
    !.
len([L], Len) :-
    len([List|], Acc, Len).

len([List|], Acc, Len) :-
    len([List|], Acc, Len).

% reversing list component
revcomp([], [ ]) :-
    !.
revcomp([[K, C]|L], [[K, C]|N]) :-
    rev(C, Cn, [ ]),
    revcomp(L, N).

rev([], Z, Z) :-
    !.
rev([X|Y], Z, W) :-
    rev(Y, Z, [X|W]).
Note added in proof

V. Chumakov and the author succeeded in finding an optimal universal tail-recursive interpreter in Theorem 5.2 with only one choice point. Thus Corollary 5.4 is strengthened as follows: For any functional logical procedure in KP an equivalent deterministic logical procedure exists in KP. So, Theorem 5.6 is strengthened significantly as follows: For any logical procedure in KDP an equivalent iterative logical procedure exists in KDP.

References