FI MU

Faculty of Informatics
Masaryk University

WiM: A Study on the Top-Down ILP Program

by

Luboš Popelínský
Olga Štěpánková

FI MU Report Series FIMU-RS-95-03

August 1995

WiM : A Study On The Top-Down ILP Program

Luboš Popelínský, Olga Štěpánková

Abstract

In the area of the inductive synthesis of logic programs it is the small number of examples which is crucial. We show that the classical MIS-like architecture can be adapted using techniques described in ILP literature so that we reach very good results if to compare with other ILP systems. We describe the top-down ILP program WiM and the results obtained through it. WiM needs from 2 to 4 examples for most of the ILP benchmark predicates. Even though it is interactive, not more that one membership query is enough to receive the correct target program. WiM has higher efficiency of learning as well as smaller dependency on the quality of the example set in comparison to some of ILP programs. The quality of learning has been tested both on good examples and on randomly chosen example sets.

1 Top-down learners

Considering interactive generate-and-test top-down learners in the context of automatic logic programming [5], four main drawbacks are being observed:

1. Too many positive examples are needed
2. The usefulness of the negative examples depends on the particular learning strategy
3. Generate (a hypothesis) and test (on the example set) strategy is too unefficient
4. Too many queries to the user are asked

We will show that even with a very small example set (less or equal to 4 positive examples) MIS-like [13] top-down learners are capable to learn most of the predicates which have been mentioned in ILP literature.

Neccessary negative examples are always dependent on the particular learning strategy and that is why it is difficult for the user to find the most appropriate ones. Our approach tries to find negative examples itself. A near-miss to one of the positive examples is considered as a candidate for that purpose. Such a negative example is found usefull if after adding that example to the current learning set, the learner is able to suggest a new definition of the target predicate. Only in such a case the user is asked for a confirmation of that particular candidate for the negative example.

In the generate-and-test strategy,it is actually the test phase – verifying the target predicate on the example set – which is too time-consuming. However, the number of hypotheses can be limited by a declaration of argument modes and by exploitation of programmer's knowledge. This knowledge can specify e.g. maximal complexity of terms, maximal number of free variables allowed in the new clause as well as maximal number of literals in a clause body. By this way we succeed to lower a cardinality of the example set, and this 'brute force' top-down learning is becoming quite efficient. The efficiency is increasing by emploing 2nd order schema for guiding the synthesis.

It is true that the query to the user is neccessary to confirm the negative example generated by the system. However, the number of queries, in general, is smaller – not more than 1 membership query is allowed – comparing to the other interactive systems [3], [13].

■

The organisation of the paper is as follows. In Section 2., we show one way how to decrease a cardinality of the search space in top-down learning algorithms. In Section 3. we revoke the assumption-based scenario as introduced in [6, 12] and than we describe WiM program. In Section 4., we review experimental results reached by WiM on good examples. We conclude by exhibiting WiM behaviour on randomly chosen examples.

■

In the folowing section, we try to estimate cardinality of the search space as a function of the size of the background knowledge and of the maximal length of clause bodies.

2 Cardinality of the search space for given settings

2.1 Upper estimate

Let BK mean the number of background knowledge predicates + 1 (for the target predicate), A the highest arity among the predicates in background knowledge and the target predicate, L the maximal length of a clause body, i.e. the maximal number of predicates in a clause body.

The number of positions of variables in a clause for a given length l is equal to a sum of the positions in the head and in the body, its upper bound is $(1+l) * A$. E.g. for `member/2` predicate, maximal length of the clause body for $l = L = 2$ and background knowledge which contains only `list(List, HeadOfList, BodyOdList)` we have

```
member(X1,X2) :- P1(X3,X4,X5),P2(X6,X7,X8)
```

i.e. 8 positions

Now we find the number $NC(n)$ of clauses for a given number n of variable positions. Having a set of variables { X1, X2, X3, X4, X5, X6, X7, X8 }, as in the example above, we can find all different clauses for fixed $P1$, $P2$. The number of those clauses is less than 8^8 because some of them are equivalent (e.g. member(X,Y) :- member(X,Z) is the same as member(U,V) :- member(U,W)). See appendix for detailed treatment of this subject. In the table bellow the values of $NC(n)$ for small values of n are displayed.

Variable positions	Clauses
1	1
2	2
3	5
4	15
5	52
6	203
7	878
8	4140
9	21147
10	115975

Table1: $NC(n)$ for small values of variable positions

As each combination of background predicates as well as the target predicate can appear in the body, we have to multiply $NC(n)$ by the number of all allowed combinations of predicate symbols. E.g. for `member/2` predicate

```
member(X1,X2) :- P1(X3,X4,X5),P2(X6,X7,X8)
```

we have 2 positions for predicates. If the maximal length of the clause body is $L = 2$ and the maximal arity $A = 3$, the number of all clauses in the search space is the sum of number of clauses of the lentgh 0 (body $==$ true), of the length 1 and 2

$$NC(A)+2*NC(2A)+3*NC(3A) = 2+2*52+3*21147 = 63547$$

The coefficients 2 and 3 are equal to the number of combinations with repetition of possible predicates in the clause body for a clause with its body length 1 and 2 respectively.

The general formula for the number of all clauses for given BK, L, A is

$$NCA = \sum_{l=0}^{L} \left(\begin{array}{c} BK+l-1 \\ l \end{array} \right) * NC((1+l)*A),$$

This formula inherits its exponential character from the function NC (see Appendix). That is why we need more information to decrease the search space. Declaration of types and a limit on the number of free variables allowed during learning the target predicate can help. It was shown elsewhere [4], that we can focus on linked clauses only. It limits the number of distinct variables significantly. We will show the way of narrowing search space in the next paragraph.

2.2 How to narrow the search space

We will demonstrate a way of narrowing search space on a simple example. Let us learn the base clause of the predicate `member/2`. The `list(List, HeadOfList, BodyOfList)` predicate is the only background knowledge predicate. Suppose we know that the maximal length of the body of the clause is 1. Then two skeletons have to be considered as candidates

(1) member(_,_)
(2) member(_,_) :- list(_,_,_)

For (1) and (2), there are $NC(2) = 2$ and $NC(5) = 52$ instances respectively, in total 54 clauses in the search space.

Let us assume that only 1 free variable may appear in the body of the clause. For the case (2) it implies introduction of 1 new variable by predicate `list/3` so that no more than 3 distinct variables are allowed. The number of clauses is than $h(1,5) + h(2,5) + h(3,5) = 41$.

If we know types of arguments, in our example `member(nom, list)`, `list(list, nom, list)`, the search space is further narrowing. In the case (1), `member(X,X)` cannot appear. The remaining `member(X,Y)` is not taken in account as it is the most general clause and it could not be consistent - any negative would be covered by this clause. Using the type limitation the case (2) may be split into

(2a) member(X,Y) :- list(L1,X,L2)
(2b) member(X,Y) :- list(L1,U,L2)

where U $\neq$ X. In (2a), as at most 1 free variable can be introduced, one of L1,L2 must be equal to Y or L1=L2. It means that only following 4 clauses remain

member(X,Y) :- list(Y,X,L1)
member(X,Y) :- list(L1,X,L1)
member(X,Y) :- list(L1,X,Y)
member(X,Y) :- list(Y,X,Y)

For (2b), as U $\neq$ X is a free variable, both L1 and L2 have to be identical to Y, and just the single clause `member(X,Y) :- list(Y,U,Y)` remains. [1]

To summarize, the search space shrinks considerably when we do exploit knowledge about the maximal length of a body of the clause, the maximal number of free variables allowed and type declarations of arguments. In the considered example of the predicate `member`, the search space consists of 5 clauses only - this compares well to the number 54 estimated in the beginning. We will show later that this way of narrowing search space is sufficient enough for a class of list processing predicates including `qsort/2`.

In the next section, we describe the *WiM* program, an offspring of the MIS-like *Markus* system [9, 10], which uses the approach described above.

3 *WiM*

3.1 Assumption-Based Learning

Ideas on an assumption-based framework underlying our methodology may be found in [2, 11]. Intuitively, in the case that a synthesizer has failed, we are looking for such a minimal extension of the given learning set which raises the chance to find a solution. The solution has to be correct and consistent with the extended example set. An assumption, if any needed, must be confirmed by a teacher.

A generic scenario consists of three parts, *inductive synthesizer*, *generator of assumptions* which generates extensions of the learning set, and *truth maintenance system* which evaluates an acceptability of both the solution found and assumptions, and which is allowed to ask teacher queries.

[1] Knowing more about the `list/3` predicate we can delete this clause, too.

3.2 Inductive synthesizer

As an inductive synthesizer, $Markus^+$ [6] has been employed, based on $Markus$ system [9, 10]. $Markus^+$ is MIS-like [13] non-interactive top-down synthesizer applying iterative deepening search in a refinement graph and controlling a search space with different parameters. It allows shifting of bias and the definition of a second-order schema which the learned program has to match.

Three parameters are used for shifting bias — the maximal number of free variables in a clause, the maximal number of goals in the body of a clause, and the maximal head argument depth ($X, [X|Y], [X, Y|Z]$, etc. are of depths 0, 1, 2, etc., respectively). $Markus^+$ starts with the minimal values of these parameters. If no acceptable result has been found, a value of one of the parameters is increased by 1 so that all variations are being tried gradually.

3.3 Generator of assumptions and truth maintenance

For each assumption which has lead to a new predicate definition, *truth maintenance system* is asking user for a confirmation/rejection of the assumption. An assumption is generated in the moment when the current example set is not complete enough so that the inductive synthesizer is not capable to find a definition of the target predicate.

As an assumption, a **near miss** to a chosen positive example is generated [6]. A near-missed example is a negative example that differs from a positive example of the intended predicate "as little as possible". A $preference\ relation$ on the set of examples is defined to generate nearmisses of less complex examples first. WiM program allows to learn predicates in two domains - lists and integers. For each of those domains the particular generator of assumptions (based of course on the same methodology) is implemented. For list processing predicates, a sum of argument lengths has been found suitable as a measure of example's complexity. For the domain of integers we use a sum of arguments. E.g. if learned `last/2` predicate from `{last(a,[a]), last(b,[c,b])}` , `last(a,[a])` is chosen.

The following syntactic approach is used for computing near misses: extend a set of constants of the chosen simpliest (preferable) example by a new constant of the correct type. Then take the preferable example and modify it by adding/deleting a list element, or by replacing an atom, using the extended constant set. In the case of integer domain arguments of the preferable example are being replaced by elements of the extended example set, by their predcessors as well as ancestors.

In our example, the constant `new` has been added so that the set of constants contains two constants, `{new, a}` and the nearmisses

```
not last(new,[a]), not last(a,[]), not last(a,[new,a]),
not last(a,[a,new])
```

are generated one-by-one. Whenever a new nearmiss has been built, it is added into the example set as the negative example and learning algorithm is called. If no solution is found that nearmiss is replaced by another one.

3.4 Basic WiM algorithm

The basic WiM algorithm is as follows:

Given

- Specification of the target predicate P (types and modes of its arguments, names and arity of background knowledge predicates to be called)
- Definitions of background knowledge predicates
- 2nd order schema of the target predicate P. P must be a specialization of the schema.
- constraints: maximal length of clauses, maximal number of free variables in the target predicate, maximal depth of arguments in a clause head
- Example set E

Algorithm:

> **(1)**
> Init bias.
> **loop**
>
> > Learn predicate P using the example set E.
> > **if** suceeded **then**
> >
> > > Simplify terms in the heads of clauses of P.
> > > Call for *truth maintenance system* to accept/reject
> > > the suggested definition P.
> > > **if** accepted **then exit**(P).

else shift bias.
if no more shift of bias **then exit(**false**)** .

pool

(2)
if (1) exited with false
loop

Generate assumption A.
if no more assumptions **then exit** false.
Add the assumption A to the learning set.
Call (1) with the extended example set $E \cup A$.
if (1) suceeded **then exit(**P**)**.
else delete the assumption A from the learning set.

pool

3.5 Term simplification in heads of clauses

WiM needs at least one uncovered positive example to introduce a new clause. It may happen that the clause is not the simpliest one. E.g. for $delete/3$ predicate and the example `delete(1,[2,1],[2])` we have got the base clause in the form

$$delete(X, [Y, X|Z], [Y|Z]) : -true$$

In this postlearning phase, all terms in all heads of clauses are being replaced by all terms which are not more complex that the original term. Only the existing variable names are allowed. For this particular clause, WiM tries to replace the term $[Y, X|Z]$ by terms of the form of $[\_, \_|\_]$, $[\_|\_]$, $\_$ and the term $[Y|Z]$ by $[\_|\_]$, $\_$ so that variables which appear in the new terms are of correct types. It means that $[Y, X|Z]$ can be replaced by $[X, Y|Z]$, $[Y|Z]$, $[X|Z], Z$ and the term $[Y|Z]$ can be replaced by $[X|Z], Z$. New candidate version of the considered clause is accepted as an adequate simplification of the original clause if after replacing the original clause by the new candidate version the target predicate definition is correct and consistent with the whole example set. If so, the more complex term is replaced and the algorithm continues until there is no term to simplify. In our example we receive

$$delete(X, [X|Z], Z) : -true$$

3.6 Minimal description length principle

The minimal description length principle can be formulated as follows [8].

> The best theory to explain a set of data is the one which minimizes the sum of
>
> - the length, in bits, of the description of the theory; and
> - the length, in bits, of data when encoded with the help of the theory.

If we consider only calls of the background knowledge predicates but not their source code, it can be easily shown that **for the predicates in Section 4. WiM will find a correct and complete solution of the minimal description length** in the given search space if a solution exists. It is still an open question whether each predicate definition reached by WiM matches the minimal description length.

4 Experimental results

WiM was examined on the following predicates:

- *member*(E, L) iff the element E appears in the list L;
- *concat*$(L1, E, L2)$ iff the list $L2$ is equal to the list $L1$ appended by the element E;
- *append*$(L1, L2, L3)$ iff the list $L3$ is equal to the list $L1$ appended by the list $L3$;
- *delete*$(E, L1, L2)$ iff the list $L2$ is the non-empty list $L1$ without its first (existing) occurrence of E;
- $reverseConcat(L1, L2)$ iff the list $L2$ has the same elements as the list $L1$ but in the reverse order. It uses *concat*$(L1, E, L2)$ predicate which appends the element E to the list $L1$;
- $reverseAppend(L1, L2)$ is the same as $reverseConcat(L1, L2)$ but using
  append$(L1, L2, L3)$;
- *last*(E, L) iff the element E is the last element of the list L;

- $split(L1, L2, L3)$ iff the lists $L2$ and $L3$ contain only odd and even elements, respectively, of the list $L1$.
- $sublist(L1, L2)$ iff the list $L1$ is a compact subsequence of the list $L2$;
- $union(S1, S2, S3)$ iff the set $S3$ is a union of sets $S1, S2$;
- $plus(I1, I2, I3)$ iff for integers $I1, I2, I3$ $I3 = I1 + I2$ holds;
- $lessOrEqual(I1, I2)$ iff for integers $I1, I2$ $I1$ is less or equal I2;
- $length(N, L)$ iff N is the length of the list L;
- $extractNth(N, L, E)$ iff E is the Nth element of the list L
- $quicksort(List, SortedList)$ iff the list $Sorted$ is has been sorted by the quicksort algorithm with $partition/4, append/3$ background knowledge.

4.1 Carefully chosen example sets

Good examples were used in the experiments described in the following two paragraphs. See Appendix B. for the used example sets.

4.1.1 Learning without assumptions

In the table bellow, we summarize the results of WiM if no assumption was needed. The first column contains the number of the needed positive examples. In the second column, there is the number of potential solutions which were tested on the example set.

	Number of positive examples	Number of hypotheses tested
member	2	3
concat	2	6
append	3	6
reverseConcat	2	5
reverseAppend	3	14
split	2	7
sublist	4	15
union	4	15
quicksort	7	2307

Table 1: WiM results if no assumption was needed

For all predicates but $quicksort/2$ we used only positive examples. CPU time was less than 5 seconds on SUN Sparc. In the case of $quicksort/2$, the example set consists of 4 positive and 3 negative examples and the learning session lasted 5 min CPU time.

4.1.2 Learning with one assumption

The following table contains the results of WiM if an assumption was generated. The contents of the table is the same as above, the final example sets consist of 2 or 3 positive examples and 1 negative example generated by the system as the assumption and afterwards verified by the user. For the complete example sets see Appendix B.

	Number of positive examples	Number of hypotheses tested
append	2	171
delete	2	21
last	2	99
plus	3	54
lessOrEqual	3	66
length	3	20
extractNth	3	27

Table 2: WiM results with 1 membership query

If generating assumptions, CPU time varied from 6 seconds - for `last/2` predicate - to 46 seconds for `append/3`.

4.1.3 Discusion of results

For predicates like `last/2, delete/2` WiM is not capable to learn the correct target definitions from positive examples only. For all predicates above, we need at worst as many positive examples as $CRUSTACEAN$ and less then $FILP$ [3] (see [5] for the comparison) with no need of negative examples. The number of queries to the user is less than for $FILP$ (see [3] for more information). MIS is not able to learn from only positive examples. $FOIL$ needs much more positive examples to succeed. Also the close world assumption employed by $FOIL$ is not appropiate for learning from a small example set.

4.2 Randomly chosen example set

In [1] a method for testing learners using randomly chosen examples has been introduced. Terms of types of lists and integers were generated randomly from a uniform distribution on structure depth 0..4. As there is a dependency betwen arguments, some of them have been derived (e.g. the first argument of *member*/2, the first argument of *sublist*/2 etc.). In opposite to [1], we don't use negative examples at all. No assumptions were generated and no interaction with user was allowed.

Following that method, we receive for WiM the following results. All couples and all triples of examples from the domain defined above were taken in account. The numbers in the table bellow means on how many couples or triples of positive examples WiM succeeds to find the correct target predicate.

$CRUSTACEAN$ needs more positive examples than WiM and generate, as a rule, more than one solution. It needs to have negative examples.

	2 positive examples	3 positive examples
member delete extractNth	64%	90%
concat reverse last split length	78%	92%
append plus lessOrEqual	-	93%

Table 3: WiM results on randomly chosen examples

The bottom-left part of the table is empty because WiM is not capable to learn those predicates from only 2 positive examples.

If we take into account that it is a human who is a source of examples, it is unlikely that s/he defines two examples of the same structure, e.g. for *member*/2 predicate `member(a,[a]), member(b,[b])` . Applying that assumption, the accuracy in the upper-left field will increase to 81%. A slight accuracy increase can be seen for the rest of table, too.

5 Conclusion

In the area of the inductive synthesis of logic programs it is the small number of examples which is crucial. We have shown that the classical MIS-like architecture can be adapted using techniques described in ILP literature so that we reach very good results if to compare with other ILP systems. We described the top-down ILP program WiM and the results obtained through it. Even though it is in principal interactive, not more that one membership query is enough to receive the correct target program. For many of predicate definitions no interaction with user is needed at all. WiM has higher efficiency of learning as well as smaller dependency on quality of the example set in comparison to several well-known ILP programs.

It would be interesting to compare WiM with the $BMWk$ methodology [7]. WiM doesn't need to have examples on the same computational chain. WiM, too, needs less examples for simple list processing predicates. However, for predicates more complex than those used as ILP benchmarks it has not been verified. Comparison with other popular ILP programs should be done, too.

We see the direction for future work mainly in weakening bias, learning multiple predicates and exploiting large background knowledge. In the current version it is the definition of argument modes which plays the significant role. However, there are some areas where it is unrealistic to ask user for mode definitions. Even the mode definitions for `member/2` predicate are litle unexpected, aren't they? Learning multiple predicates is a very challenge. Actually the ancient MIS could that. Background knowledge(BK) predicates , as for most (if not all) ILP programs are always well-chosen, i.e. BK consists of almost only the needed predicates. In real-world applications it looks little different.
It is still an open question whether and under what condition each predicate definition reached by WiM matches the minimal description length.

Acknowledgments

We would like to thank Pierre Flener, Norbert Fuchs, Alípio Jorge and Guillaume LeBlanc for stimulating discussions and the anonymous referee for his suggestions. Thanks for comments on the early version of this paper are due to Pavel Brázdil. The significant part of this work was done during a stay of the first author in Porto, thanks to TEMPUS IM grant.

References

[1] Aha D.W., Lapointe S., Ling C.X., and Matwin S.: Inverting implication with small training sets. In Bergadano F., De Raedt L. (Eds.) Proc. of ECML'94, Catania, LNCS 784, pp. 31–48, Springer Verlag 1994.

[2] Bondarenko A., Toni F., and Kowalski R.A.: An assumption-based framework for non-monotonic reasoning. In Perreira L.M., Nerode A. (Eds.) Proc. of the 2nd Int'l Workshop on Logic Programming and Non-Monotonic Reasoning, Lisbon, 1993, pp. 171–189, MIT Press, 1993.

[3] Bergadano F. and Gunetti D.: An interactive system to learn functional logic programs. *Proc. of IJCAI'93*, Chambéry, pp. 1044–1049.

[4] De Raedt, L.: Interactive Concept-Learning. PhD Thesis, Catholic University Leuven, Belgium 1991.

[5] Flener P., Popelínský L.: On the use of inductive reasoning in program synthesis: Prejudice and prospects. Proc. of the 4th Int'l Workshop on Logic Program Synthesis and Transformation (LOPSTR'94), Pisa, Italy, 1994.

[6] Flener P., Popelínský L. Štěpánková O.: ILP nad Automatic Programming: Towards three approaches. Proc. of 4th Workshop on Inductive Logic Programming (ILP'94), Bad Honeff, Germany, 1994.

[7] Le Blanc G.: BMWk Revisited. In Bergadano F., De Raedt L. (eds): *Proc. of ECML'94*, Catania, pages 183-197. LNCS 784, Springer Verlag, 1994.

[8] Li M., Vitanyi P.: An Introduction to Kolmogorov Complexity And Its Applications. Springer Verlag New York 1993.

[9] Grobelnik M.: Induction of Prolog programs with Markus. In Deville Y.(ed.) Proceedings of LOPSTR'93. Workshops in Computing Series, pages 57-63,Springer-Verlag, 1994.

[10] Grobelnik M.: Declarative Bias in Markus ILP system. *Working notes of the ECML'94 Workshop on Declarative Bias*, Catania, 1994. (chairperson Rouveirol C.)

[11] Kakas A.C., Kowalski R.A., and Toni F.: Abductive logic programming. Journal of Logic and Computation 2, 6, pp. 719-770, 1992.

[12] Popelínský L.: Towards Program Synthesis From A Small Example Set. In: Proceedings of 10th Workhop on Logic Programming WLP'94, Zurich, Switzerland, 1994.

[13] Shapiro Y.: Algorithmic Program Debugging. MIT Press, 1983.

Appendix A: Number of admissible sequences of variables

A sequence of variables $\{X_1, ..., X_i\}$ is **admissible** if no variable X_{j+1} can appear before all variables $\{X_1, ..., X_j\}$ have been used.

In order to count the number of admissible sequences of variables of a given length, it is useful to introduce a function $h(P, N)$. This function specifies the exact number of those admissible sequences of variables of the length N in which just P variables appear. Obviously, this function is defined only for $P \leq N$ (all P variables have to be present in the considered sequence). This function is easy to evaluate for distinguished values of its arguments, namely

$$h(1, N) = 1$$
$$h(P, P) = 1$$

Number of admissible seguences of the length N with just 2 variables is given as a sum of cardinalities of those sets of admissible sequences which differ by the positions of the first occurence of X_2. Variable X_2 can appear first on the position 2, and then on all higher positions, i.e.

$$h(2, N) = 2^{N-2} + 2^{N-3} + ... + 1 = 2^{N-1} - 1$$

For other values of its arguments the function h can be defined recursively as follows

$$h(P, N) = P * h(P, N-1) + h(P-1, N-1).$$

The number $NC(N)$ of all admissible sequences of variables of the length N is then given as a sum

$$NC(N) = h(1, N) + h(2, N) + h(3, N) + ... + h(N-1, N) + h(N, N).$$

and the number of all sequences of K variables of the length N is given as a sum

$$NC(K, N) = h(1, N) + h(2, N) + ... + h(K, N).$$

Obviously, for $N > 1$ there holds

$$NC(N) > h(2, N) + 1 = 2^{N-1}$$

The function $NC(N)$ has clearly an exponential character.

Appendix B: Example sets

Definitions of predicates in the form of

```
  pred_def( Predicate/Arity, <arguments types and modes>,
      <background knowledge predicate to use> , []).
```

and examples of the given predicate follows.

```
pred_def( member/2, [ -x, +xl ], [ member/2 ], [] ).
ex( member(a,[a]),true).
ex( member(c,[b,c]),true).

pred_def( concat/3, [+xl, +x, -xl], [concat/3], []).
ex( conc([],a,[a]), true).
ex( conc([b],c,[b,c]), true).
ex( conc([b,c],d,[b,c,d]), true).

pred_def( append/3, [ +xl, +xl, -xl ], [ append/3 ], [] ).
ex( append( [], [ a ], [ a ] ), true ).
ex( append( [  b , c], [  d , e], [ b, c, d, e ] ), true ).
ex( append( [ f ], [ g , h ], [f ,g, h ] ), true ).

pred_def( delete/3, [ +x, +xl, -xl ], [delete/3], [] ).
ex( delete(1,[2,1],[2]),true).
ex( delete(3,[4,5,3,6],[4,5,6]),true).
```

Assumption: delete(2,[2,1],[2]),false

```
pred_def( reverseConcat/2, [ +xl, -xl ], [ cconc/3, reverseConcat/2 ],
[] ).
ex( reverseConcat([], []), true).
ex( reverseConcat( [ a, b, c ], [ c, b, a ] ), true ).

pred_def( reverseAppend/2, [ +xl, -xl ],
[ ssingleton/2, append/3, reverseAppend/2  ], [] ).
```

```
ex( reverseAppend([1,3,4],[4,3,1]), true ).
ex( reverseAppend([2,0],[0,2]),true ).
ex( reverseAppend( [], []), true).

pred_def( last/2, [-x, +xl], [last/2], []).
ex( last(a,[a]),true).
ex( last(b,[a,b]),true).
```
Assumption: last(a,[a,b]),false

```
pred_def(split/3, [+xl, -xl, -xl], [split/3], []).
ex( split([x,y], [x], [y]), true).
ex( split([1,2,3,4], [1,3], [2,4]), true).

pred_def( sublist/2, [ -xl, +xl ], [ sublist/2 ], [] ).
ex( sublist([],[]),true).
ex( sublist( [ c, d ], [ b, c, d, a ] ), true ).
ex( sublist( [ c, d ], [ c, d, b, a ] ), true ).
ex( sublist( [ a ], [ b, a]), true).

pred_def( union/3, [ +xl, +xl, -xl ], [ member/2, union/3 ], [] ).
ex( union([],[1,2,3],[1,2,3]), true).
ex( union([1,3],[2,3,4],[1,2,3,4]), true).
ex( union([1,2,3,4],[2,3,5],[1,4,2,3,5]), true).
ex( union([1,2,3],[3,4,5],[1,2,3,4,5]), true).

pred_def( plus/3, [+int, +int, -int], [plus/3], []).
ex( plus(0,s(0),s(0)), true ).
ex( plus(s(s(0)),s(s(0)),s(s(s(s(0))))), true ).
ex( plus(s(0),s(s(0)),s(s(s(0)))), true ).
```
Assumption: plus(0,0,s(0)), false

```
pred_def( leq/2, [+int, +int], [leq/2], [] ).
ex( leq(0,s(0)), true ).
ex( leq(s(s(s(0))),s(s(s(s(0))))), true ).
ex( leq(s(s(0)),s(s(s(0)))), true ).
```
Assumption: leq(s(s(s(s(0)))),s(0)) , false

```
pred_def( length/2, [ +xl, -int ], [ length/2, is0/1, ppl/3 ], [] ).
ex( length( [], 0 ), true ).
ex( length( [ b, c ], s(s(0)) ), true ).
ex( length( [ f ], s(0) ), true ).
```

Assumption: length([0],0) , false

```
pred_def(extractNth/3, [-int, +xl, -xl], [extractNth/3], [] ).
ex( extractNth( s(0), [ s(s(s(0))) ], []), true).
ex( extractNth( s(0), [ s(s(0)), s(0), (s(s(0))) ], [s(0), (s(s(0))) ]),
    true).
ex( extractNth( s(s(0)), [  s(s(0)), s(0), (s(s(0)))  ],
    [ s(s(0)), (s(s(0)))  ]), true).
```

Assumption: extractNth(s(s(s(0))),[s(s(s(0)))],[]) , false

Publications in the FI MU Report Series are in general accessible via WWW and anonymous FTP:

```
http://www.fi.muni.cz/informatics/reports/
ftp  ftp.fi.muni.cz (cd pub/reports)
```

Copies may be also obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic