AN INTERPRETER FOR THE PROGRAMMING LANGUAGE PREDICATE LOGIC

Sten-Ake Tarnlund
Computer Science Department
University of Stockholm

Abstract

We describe an Interpreter for the programming language predicate logic. Some topics are; syntax and proof procedure, procedure evocation, function transformation, goal variation and interactive computational control.

Introduction

The development of programming languages has more and more relieved a programmer from machine control and details, giving him the power of abstract reasoning and thus helped him to concentrate more on his problem. Kowalski [8,9] has convincingly argued that predicate logic formalizes a man's thoughts and is a useful simple programming language for human problem solving. Consequently It is desirable to have predicate logic implemented on machines for practical programming. There is one implementation at the University of Aix-Marseille, PROLOGUE [2]. At the University of Stockholm we have implemented an interpreter in LISP 1,6 [11].

Syntax and Proof Procedure

The syntax of the language consists of Kowalski's procedural form [7] plus a control structure consisting of an ordered procedure, ordered procedures and computational consequences (Tarnlund [14]). The proof procedure is based on Kowalski's connection graph system (6].

Kowalski's procedure form

A procedure is the only type of statement in the language. It contains two special cases, a goal and an assertion. Two typical procedures are

$$Own(x,y) \clubsuit Own(x,z)_t Part(y,z),$$

which is read "x owns y if there Is a z such that x owns z and y is a part of z", and

$$D(x,y),D(x,z) * D(x_t w),P(x),M(y,z,w)_t$$

which is read "x divides y or x divides z if there is a w such that x divides w and x is a prime and y - z * w". The part to the left of the arrow is called procedure name and the part to the right is called procedure body consisting of procedure calls.

A goal statement is a procedure without a name e.g.,

$$\vdash Own(John,tyre).$$

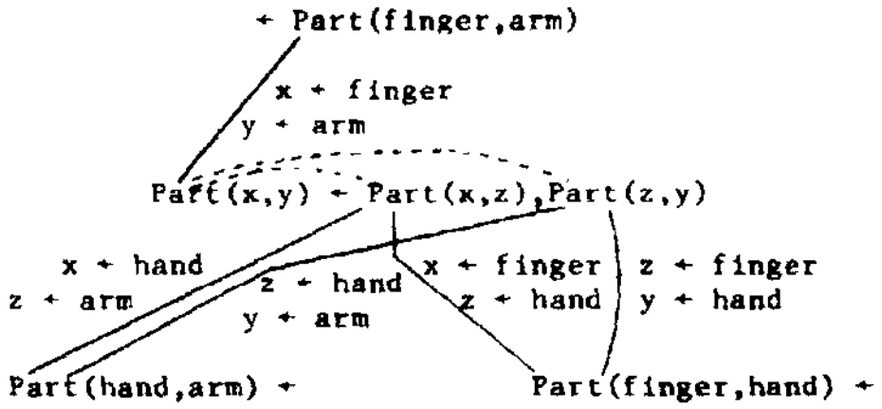
An assertion Is a procedure without a body e.g. ,

$$Part(tyre,car) \ll .$$

A program is a set of procedures connected In a graph depending on how they match each other. For example, the program

```
    ← Part(finger,arm)
      Part(x,y) ← Part(x,z),Part(z,y)
      Part(hand,arm) ←
      Part(finger,hand) ← ,
```

gives rise to the following connection graph



Flg.l. A connection graph. Each atom is connected to all other atoms with the same name occurring on opposite sides of the arrows, provided that they can be matched i.e., unified (12). Recursive procedures are linked with dotted lines.

The connection graph is indipendent of the order the programmer presents his statements. Depending on one's programming style (problem-solving) one can combine top-down, middle-out and bottom-up programming. The goal of the interpreter is to activate links (resolve corresponding atoms) in a way that eventually leads to the empty procedure D .

Control structure

The control structure transfers some decision-making responsibility from the search strategy to the programmer and makes the decisions explicit. This is illustrated by a program that probes for a node to be deleted in a binary tree such that its remaining nodes are sorted when traversed in symmetrical order. The program is based on a delete algorithm in Knuth [5]. We use Knuth's zodiac example (sec. 6.2.2) to delete the node CAPRICORN from a binary tree. The initial goal is

$$\vdash Delete^*(CAPRICORN). \qquad (1)$$

Suppose that we have the binary tree in a data base

$$A(tree) \vdash , \qquad (2)$$

where A is an arbitrary predicate whose argument tree is equal to

t(t(0,AQUARIUS,t(O.ARIES,t(0,CANCER,0))), (3)
CAPRICORN,t(t(0,GEMINI,t(0,LEO,t(0,LIBRA,0))),
PISCES,t(t(0,SCORPIO,0),TAURUS,t(O.VIRCO.O)))),

This zodiac binary tree "tree" Is an Instantiation of the data structure

t(x,y,z)

denoting a binary tree with a root y and a left subtree x and a right subtree z. An empty tree Is denoted 0. When we have picked up the binary tree from the data base and deleted the node CAPRICORN, we want to put the new tree back into the data base i.e.,

(A(z) * },Delete*(u) - <A(x),Delete(x,u,z)>, (4)

where u is the node to be deleted in a binary tree x, and z is the resulting binary tree. Note that Delete and Delete are two different predicates. The predicate within the braces is a computational consequence not to be resolved upon during the computation, but to be put into the data base when the remaining part of procedure (4) is resolved to the empty clause. (Computational consequences may be viewed as a generalization of Green[f]s answer predicate [3]). The angled brackets '<' and '>' tell us that procedure (4) is an ordered procedure and is executed from left to right within the brackets.

In the delete algorithm we check each node in the binary tree to determine whether it is to be deleted or if we have to continue a binary tree search for the node to be deleted. The procedures for this algorithm are ordered among themselves and delimited by a pair of double angled brackets '<<' and '>>'. This ordering controls the execution of the procedures.

There are two cases when a node is deleted. The simpler case occurs when the left subtree of the right subtree of a node u is empty viz.,

« Delete(t(x,u,t(0,y',z')),u,t(x,y\z')) + , (5)

where the new binary tree

t(x,y\z')

is "the old tree with the root u deleted". The control structure attempts to execute this procedure first. In case we have to search for the successor node y of the deleted node u in symmetrical order, the procedure is more complicated viz.,

Delete (t (x,u, t (x \ y \z »)) ,u, t (x,y , t (x",y \z ')) -
Successor(x',y,x"), (6)

where

t(x,y,t(x",y\z'))

is the new binary tree in which the root y and the left subtree x" are still unknown. They are deduced by the Successor procedures (9) and (10).

In case we have not found the node to delete we have to search for it viz.,

Delete(t(x,y,z),u,t(x',y,z)) *
<LE(u,y),Delete(x,u,x[f])>, (7)

where the node u to be deleted is to be found in the left subtree x of y. The new binary tree becomes

t(x[l],y,z),

where x[f] is the same tree as x but with a node deleted. When the probed node is in the right subtree of y we have

Delete(t(x,y,z),u,t(x,y,z')) ←
<Gr(u,y),Delete(z,u,z')> >>, (8)

where the node u to be deleted is found in the right subtree z. LE and Gr are "built-in" procedures that determine alphabetic orderings (see below). The procedures (5) - (8) are ordered among themselves, which means that a procedure call Delete(x,y,z) first will try procedure (5); if that procedure cannot be resolved to the empty procedure the call will try procedure (6) and so on.

To complete the program we write the recursive successor procedures that probe for a node y which is the successor in symmetric order of the deleted node u v i z .,

<< Successor(t(0,y,z),y,z) ← (9)

and

Successor(t(x,y,z),y',t(x',y,z)) ←
Successor(x,y',x') >>, (10)

where (10) is the recursion step and (9) the recursion base which terminates the search for the successor node on the first root with an empty left subtree.

Procedures (5), (6), (9) and (10) with the goal

+ Delete(tree,CAPRICORN,z),

where tree is equal to the binary tree in (3), have been executed on the theorem prover of Allen & Luckham [1], at the Artificial Intelligence Laboratory, Stanford University, in 2.6 seconds (see appendix).

Procedure Evocation

A computation is mainly goal oriented i.e., the interpreter tries to find out a deduction top-down from the goal. There are, however, some important deviations.

Bottom-up pre-processing

It is generally recognized that it is important to find the "right" data structure for a problem *and to make strong use of it. For* predicate logic programs this can mean improvements of several magnitudes. The reason for such improvements is that predicate logic programs can be very nondeterministic but by making a bottom-up pre-processing of the data structure the program can become much more deterministic. We show this idea with a slight variation of an example from Kowalski [9].

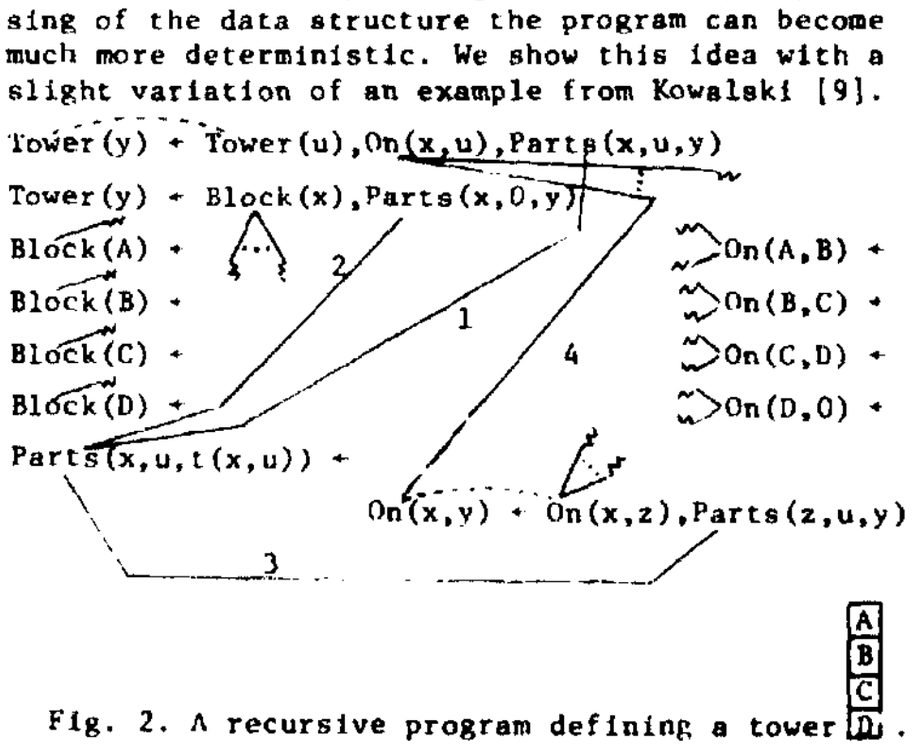


Fig. 2. A recursive program defining a tower .

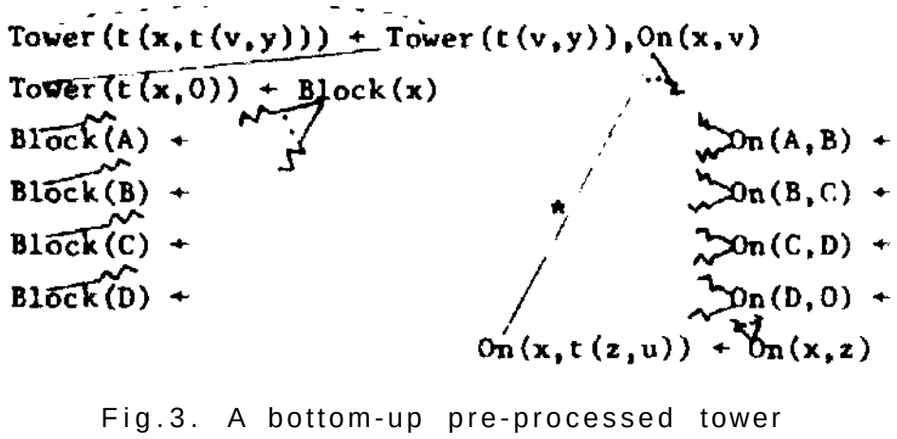


Fig.3. A bottom-up pre-processed tower program.

The program in fig.3 with the data structure pre-processed into the procedures seems to use the "right" data structure t(x,u) for this problem where x is a top block and u is either an empty tower or a tower t(v.y) where y is an empty tower, denoted 0, or a non-empty tower. Notably, this more deterministic tower program is more than 15 times faster than the program in fig.2 for the goal

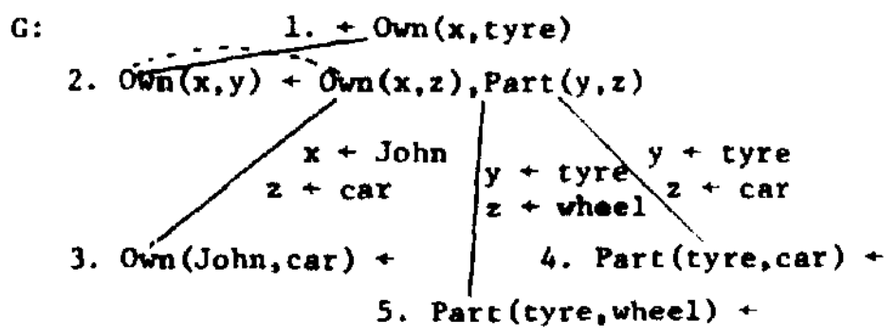
← Tower(t(A,t(B,t(C,t(D,0))))).

when run on the theorem-prover at the Artificial Intelligence Laboratory, Stanford University (see appendix).

In a top-down execution the variable v of atom On(x,v) will always be bound to a block so the starred link in fig.3 will never be activated. Consequently we can erase it, which then implies that the recursive "On" procedure contains a pure literal and thus can be deleted from the graph. This reasoning about data types would be simplified if the variables and their types were declared. Most of the pre-processing that we have seen follow a one link rule (see below).

One link evocation

Another important deviation from a top-down computation can occur from a one link evocation. The one link rule, however, is also useful during a top-down computation, because of the fact that a whole procedure can be deleted from a graph, if the procedure contains an atom connected with only one link, which is activated (Kowalski [6]). The following figures show a one link evocation.



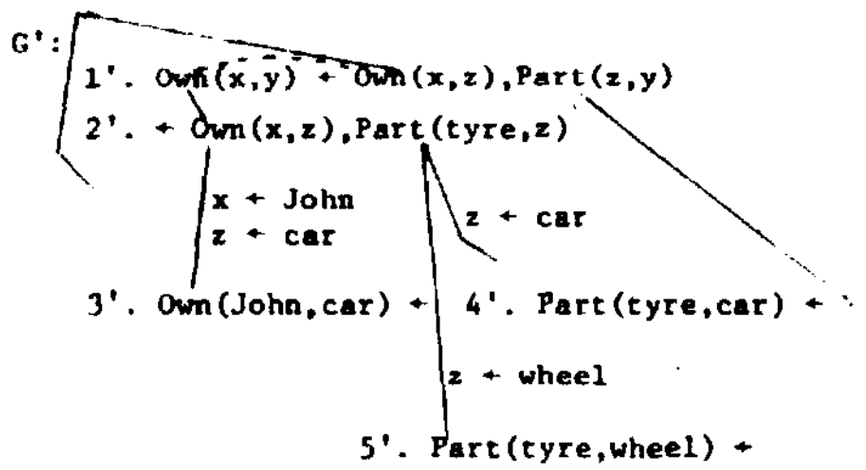


Fig.4. Two connection graphs G and G'. Graph G'. follows from G by evoking statement 1 in a top-down computation. Interesting substitutions are shown. The dotted links show recursion.

There are two atoms connected with only one link in graph G. Selecting statement 1 gives a top-down computation and a new goal

← Own(x,z),Part(tyre,z)

in C. This goal is non-deterministic in the sense that the interpreter is not told which subgoal to solve first. This problem is discussed tjelow in the sections built-in procedures and instantiation evocation. The old goal

← Own(x,tyre)

In G, is deleted giving a cleaned up graph G'.

The one link rule does not work in G'; however, the interpreter could have followed the one link rule in a bottom-up computation by evoking statement 3 giving G".

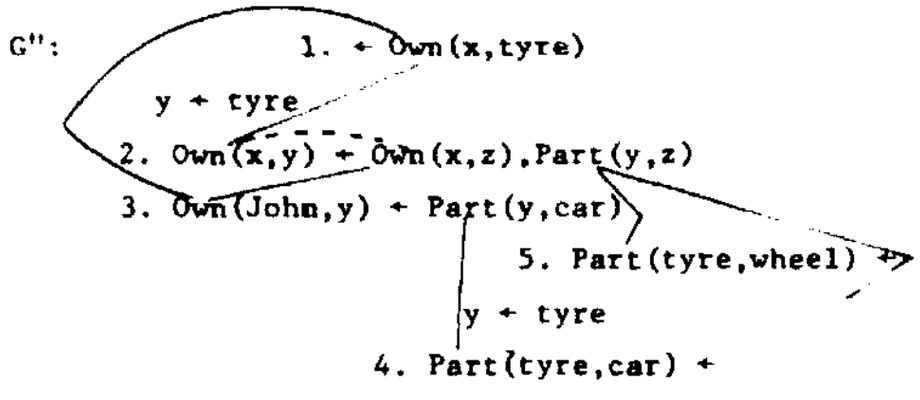


Fig.5. Graph G" follows from G by evoking statement 3 In a bottom-up computation.

The one link rule is applicable again in G", this time middle-out, giving the new fact

Own(John.tyre) ← ,

which is a recursion base. The problem is now solved top-down using the recursion base giving a solution x ← John in the goal

← Own(x,tyre).

The transformation from C to C" shows an efficient problem-solving sequence by the one link rule combining bottom-up, middle-out and top-down computation. Bottom-up and middle-out execution corresponds to PLANNER'S (Hewitt [A]) antecedent theorem and top-down to a consequent theorem.

A matcher containing a unification algorithm (Robinson [12]) plays a major role, operating on the substitution sets corresponding to a link. The matcher computes the substitution set to a new link in a local area.

Instantiation evocation

The graph G^1 in fig.4 has no one link atom, however, in the goal

Own(x.z) .Part (tyre,z)

the second subgoal is more instantiated than the other. Tliis fact is a heuristic giving the interpreter a hint that a first attack on that goal could give a more deterministic computation (Mlnker et al. (10]). The interpreter recognizes also that the subgoals have common variables i.e., they are dependent. So when searching for solutions for z tn Part(tyre,z), the Interpreter checks these for compatibility with the subgoal

<--Own(x.z).

Built-in procedures

The interpreter also has built-in knowledge that makes it simpler to write programs. We are nlready familiar with the built-in predicates l.K(u,y) and Gr(u,y) (the delete program above) which arc true when the relations u < y and y > u are satisfied.

We can also use ordinary numbers like 0, 1, 2, ... instead of 0, s(0), s(s(0)), ... , where s Is a successor function.

The interpreter has a knowledge base of useful LISP-functions [11] which can be used as procedures. The knowledge base can be increased by a programmer. Some useful LISP-functions e.g., add, difference and cons which can be represented as the following procedures: ADD(x,y,z), DlFF(x,y,z) and C0NS(x,y,z), where x and y are input variables and z is an output variable. If the input variables are instantiated with numbers then the ADD and DIFF predicates are solved directly e.g.,

ADD(2,5,z)

is solved with z 7. The procedure CONS(x,y,z) is even solved when the output variable z is bound to a list e.g.,

CONS(x,y,[a,b,c])

gives x + a and y <--- [b,c]

The matcher can solve more complex problems like the three dependent subproblems

+- ADD(l,y,z),ADD(x,2,z),ADD(x,y,3)

from Its knowledge base in LISP-code. The solution is equivalent to the following sequence of substitutions, y <--- diff[z,l], and x <--- diff[z,2] giving addf difflz,2],diff[z,l]]=3,whichgives z-<---3, y <--- 2 and x <-- 1.

Built-in procedures like LE and Gr are used frequently at decision points during a computation thus they are given high priority to be matched.

Function Transformation

In the section on pre-processing predicates were Instantiated with a data structure (function). Sometimes, however, we want to do the opposite i.e., move a function out of a predicate and transform it to a predicate. We explain this idea from Kowalski [9] with an example.

Suppose that we have to sort a list of three elements [c,b,a] i.e., the initial goal is

<-- Sort(cons(c,cons(b,cons(a,nil))),z), (1)

where z is the output variable and x is the first element and y the rest of the list cons(x.y). Assume further that we have a procedure

Sort(list(x),z) <---0(list(x)) (2)

in the sort program. Procedures (1) and (2) do not unify though their first arguments denote a list. But let us transform the term 'list' in (2) to a predicate and also use two auxiliary procedures i.e.,

Sort(u.z) <--- 0(x),List(u,x) (3)

and

List(cons(x',y'),list(x',u'))<----List(y',u') (4)

List(cons(x',nil),list(x'.nil)) (5)

then the situation is changed and yields after some computational steps the intended goal

+ 0(list(c.list(b,list(a,nil)))). (6)

The transformation of a term into a predicate extends the domain of the unification algorithm and gives the opportunity to use a uniform procedure for terms and predicates to determine which of them to evoke.

Goal Variations

In problem-solving problem variation sometimes leads to a successful solution (Polya [13]). We show two ways to vary a problem I.e., problem generalization and problem specialization. In both cases logical consequence plays an important role.

Problem generalization

Perhaps the simpliest example that shows both problem generalization and specialization is a family example from Kowalski [9].

Suppose that we have the following goal

*- Male(x),Parent(x,y),Father(x)

and the procedure

Father(x) * Male(x),Parent(x,y),

then by logical consequence we have the new goal

<---Male(x),Parent(x,y).

The problem is generalized in the way that the connection graph for the new goal contains less details in form of predlcate(s) and link(s), yet still gives the same solution.

Problem specialization

A reasoning by logical consequence can also give a specialized problem in the way that we put in more details in form of link(s) and predicate(s) into the connection graph.

Suppose we have the goal

<----Male(x),Parent(x,y)

and the procedure

Father(x)<---Male(x),Parent(x,y),

then by logical consequence we have the new goal

<---Male(x).Parent(x,y).Father(x).

The idea of logical consequence can also be used in a heuristic reasoning.

Suppose that we have a goal to show that x is a parallelogram

← Parallelogram(x) (1)

and a procedure defining a parallelogram

Parallelogram(x) ← **Plane(x),Sides(u,v,w,y,x),**
Parallel(u,v),Parallel(w,y),
Opposite(u,v),Opposite(w,y) (2)

and two rectangle procedures

Rectangle(x) ← Parallelogram(x),Rightangled(x), (3)

Rectangle(A) ← . (4)

Let us assume that the goal we obtain by resolving (1) and (2) Is very non-deterministic; then we could use the idea of logical consequence in a heuristic reasoning and make the following assumption.

Suppose that Rectangle(x) is a logical consequence of Parallelogram(x), then we have the new specialized goal,

← Parallelogram(x),Rectangle(x), (5)

which by resolution, (A) and (5), gives the goal

← Parallelogram(A), (6)

which is more deterministic and could give a faster solution. This computation is, however, based on a heuristic assumption that could be false and thus could be a wasted computation.

Whether or not It Is favourable to generalize or specialize a problem depends basically on how much each can decrease the non-determinism of a problem.

Interactive Computation Control

Interactive control of a computation helps a programmer debug a program, as well as improve a program, and can give the programmer hints that he can use to trim the search strategy.

Program debugging

A computation is a deduction (a unique property of predicate logic among programming languages), which means that each step In the computation is a logical step. This gives us a powerful mean to discover bugs e.g., misspellings, missed termination conditions (recursion bases) and badly expressed ideas.

Program Improvements

Assume that we have a data base holding

$A(k_1,1) \leftarrow$, $A(k_2,2) \leftarrow$,..., $A(k_n,n) \leftarrow$ (1)

and that we want to search in the data base for a special key k

← A(k,z), (2)

which a priori can be matched with all elements in the data base, then this program is reasonable for very few elements only. If a programmer is told when the number of possible matches is more than a given threshold then he could start thinking of some program Improvements.

A simple refinement of the given program is to substitute goal (2) by goal

← A(k,1) (3)

and add the procedure

A(x,y) ← A(x,y+1), (A)

which gives a deterministic sequential search program.

Conclusions

Our experience in predicate logic programming has only been positive. A program becomes structured and transparent. All variables of a procedure are local variables of that procedure so we do not get side effects from global variables. Each step in a computation is a logical step, which supports debugging. A proof of an algorithm on a theorem-prover has a different behaviour from, for example a proof in group theory where a proof of depth 9 is a surprise, though a proof of depth 42 for an algorithm has been found in 75 seconds (Ta'rnlund

Acknowledgements

The design of the interpreter is very much inspired by Robert Kowalski of Imperial College who visited University of Stockholm in December 74. James McSkimin of University of Maryland and Ake llansson of University of Stockholm have given many valuable comments on drafts of this paper. David Luckham, Jorge Morales and Joachim Schreiber of Stanford University made the appendix possible.

NFR (Naturvetenskapliga Forskningsradet) and ITM (Institutet for Tillampad Matematik) have partly supported this work.

References

1. Allen, J., and Luckham, D., An Interactive Theorem-Proving Program, Machine Intelligence 5, American Elsevier Publishing Company, New York 1970.
2. Colmerauer, A., Kanoul, H., Pasero, R., and Roussel, P., Un Systeme de Communication Homme-Machine en Francais, Rapport Preliminaire, Groupe de Researche en Intelligence Artificielle Universite d'Aix-Marseille, Luminy 1972.
3. Green, C, The Application of Theorem Proving to Question Answering Systems, Doctorial dissertation, Electrical Engineering Department, Stanford University 1969.
4. Hewitt, C., Description and Theoretical Analysis (using schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. Al Memo No 251, MIT, Project MAC, 1972.
5. Knuth, D., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley 1973.
6. Kowalski, R., An Improved Theorem-proving System for First-order logic, D.C.L. Memo No 65, University of Edinburgh 1973.
7. Kowalski, R., Predicate Logic as Programming Language, Proceedings of IFIP 1974.
8. Kowalski, R., Logic for Problem Solving, Memo No 75, University of Edinburgh 1974.
9. Kowalski, R., Oral communication.
10. Minker, J., Fishman, D.H., and McSkimin, J.R., The Q* algorithm, Artificial Intelligence Nn 3, 1973.

11. Quam, L,H. and Diffie, W., Stanford LISP 1.6 Manual, AI Operating Note 28.7, Stanford University 1972.

12. Robinson, J.A., A Machine-oriented Logic Based on the Resolution Principle, Journal of the Association for Computing Machinery, Vol. 12, 1965.

13. Polya, G., How to Solve I t , 2d. ed., Doubleday & Company, Inc., Garden City, N.Y., 1957,

14. Tarnlund, S-A., On Logic Programming, Computer Science Department, University of Stockholm 1975.

APPENDIX

We show three programs presented in the text. The proofs of the programs are from the theorem-prover of Allen & Luckham [1] of Stanford University.

A DELETE PROGRAM

The program deletes a node in a binary tree. In the showed computation the node CAPRICORN is deleted from the binary tree that perserves the symmetrical order among the remaining nodes. The original tree is the first argument and the new tree is the second in the predicate DELETE (bottom) given by the answer extractor.

```
PRE_PRED:DELETE,SUCCESSOR;

VAR:x,y,z,y1,y2;

PRE_OP:0,t,AQUARIUS,ARIES,CANCER,CAPRICORN,PISCES,GEMINI,LEO,LIBRA,
TAURUS,SCORPIO,VIRGO;

DELETE(t(x,y2,t(0,y1,z1)),y2,t(x,y1,z1));

SUCCESSOR(x1,y,x2)⊃DELETE(t(x,x3,t(x1,y1,z1)),x3,t(x,y,t(x2,y1,z1)));

SUCCESSOR(t(0,y,z),y,z);

SUCCESSOR(x,y1,x1)⊃SUCCESSOR(t(x,y,z),y1,t(x1,y,z));
THM
∃z DELETE(t(t(0,AQUARIUS,t(0,ARIES,t(0,CANCER,0))),CAPRICORN,t(t(0,GE~
MINI,t(0,LEO,t(0,LIBRA,0))),PISCES,t(t(0,SCORPIO,0),TAURUS,t(0,VIRGO,~
0)))),CAPRICORN,z);

HERE-ARE-THE-CLAUSES:
2 DELETE(t(x,y,t(0,z,y1)),y,t(x,z,y1));
3 SUCCESSOR(z,x1,x2)⊃DELETE(t(x,y,t(z,y1,y2)),y,t(x,x1,t(x2,y1,y2)));
4 SUCCESSOR(t(0,x,y),x,y);
5 SUCCESSOR(x,y1,y2)⊃SUCCESSOR(t(x,y,z),y1,t(y2,y,z));
6 ~DELETE(t(t(0,AQUARIUS,t(0,ARIES,t(0,CANCER,0))),CAPRICORN,t(t(0,GE~
MINI,t(0,LEO,t(0,LIBRA,0))),PISCES,t(t(0,SCORPIO,0),TAURUS,t(0,VIRGO,~
0)))),CAPRICORN,x);
7 ~SUCCESSOR(t(0,GEMINI,t(0,LEO,t(0,LIBRA,0))),x,y);
COUNT
2
LEVEL
1
ELAPSED-TIME
2617
8 ~SUCCESSOR(0,x,y);
NIL 1 2
1 ~SUCCESSOR(t(0,GEMINI,t(0,LEO,t(0,LIBRA,0))),x,y):3 4
2 SUCCESSOR(t(0,x,y1),x,y);AXIOM
3 SUCCESSOR(z,x1,x2)⊃DELETE(t(x,y,t(z,y1,y2)),y,t(x,x1,t(x2,y1,y2)));~
AXIOM
4 ~DELETE(t(t(0,AQUARIUS,t(0,ARIES,t(0,CANCER,0))),CAPRICORN,t(t(0,GE~
MINI,t(0,LEO,t(0,LIBRA,0))),PISCES,t(t(0,SCORPIO,0),TAURUS,t(0,VIRGO,~
0)))),CAPRICORN,x);THM

ANSWER:
(DELETE(t(t(t(0,AQUARIUS,t(0,ARIES,t(0,CANCER,0))),CAPRICORN,t(t(0,GEMI~
NI,t(0,LEO,t(0,LIBRA,0))),PISCES,t(t(0,SCORPIO,0),TAURUS,t(0,VIRGO,0)~
))),CAPRICORN,t(t(0,AQUARIUS,t(0,ARIES,t(0,CANCER,0))),GEMINI,t(t(0,L~
EO,t(0,LIBRA,0)),PISCES,t(t(0,SCORPIO,0),TAURUS,t(0,VIRGO,0))))))
```

TOWER PROGRAM 1

This tower program does not use any data structure exept for the PARTS assertion, which makes it leas efficient than program 2.

```
VAR: u,x,v,y,z;

PRE_PRED: PARTS,TOWER,ON,BLOCK;

PRE_OP: t,A,B,C,D,0;

PARTS(x,u,y)∧ON(x,u)∧TOWER(u)⊃TOWER(y);

PARTS(x,0,y)∧BLOCK(x)⊃TOWER(y);

PARTS(x,u,t(x,u));

PARTS(z,u,y)∧ON(x,z)⊃ON(x,y);

ON(A,B); ON(B,C); ON(C,D); ON(D,0); BLOCK(A); BLOCK(B); BLOCK(C); BLOCK(D);

THM:

TOWER(t(A,t(B,t(C,t(D,0)))));;


CHOICE-STRATEGY-IS:
VINE∧SUPPORT(THM);

EDIT-STRATEGY-IS:
DEPTH(6)∨LENGTH(3);

ELAPSED-TIME =74250

NIL 1 26
1 ¬(ON(C,u)∧PARTS(u,x,t(D,0)));3 24
3 ¬ON(C,t(D,0));5 26
5 ¬(BLOCK(u)∧(ON(C,t(D,0))∧PARTS(u,0,t(D,0))));7 8
7 ¬(ON(C,t(D,0))∧TOWER(t(D,0)));9 26
8 BLOCK(x)∧PARTS(x,0,u)⊃TOWER(u);AXIOM
9 ¬(ON(u,x)∧(TOWER(x)∧PARTS(u,x,t(C,t(D,0)))));11 28
11 ¬TOWER(t(C,t(D,0)));13 26
13 ¬(ON(B,u)∧(TOWER(t(C,t(D,0)))∧PARTS(u,x,t(C,t(D,0)))));15 24
15 ¬(ON(B,t(C,t(D,0)))∧TOWER(t(C,t(D,0))));17 26
17 ¬(ON(u,x)∧(TOWER(x)∧PARTS(u,x,t(B,t(C,t(D,0))))));19 28
19 ¬TOWER(t(B,t(C,t(D,0))));21 26
21 ¬(ON(A,u)∧(TOWER(t(B,t(C,t(D,0))))∧PARTS(u,x,t(B,t(C,t(D,0))))));23 24
23 ¬(ON(A,t(B,t(C,t(D,0))))∧TOWER(t(B,t(C,t(D,0)))));25 26
24 ON(u,v)∧PARTS(v,y,x)⊃ON(u,x);AXIOM
25 ¬(ON(u,x)∧(TOWER(x)∧PARTS(u,x,t(A,t(B,t(C,t(D,0)))))));27 28
26 PARTS(u,x,t(u,x));AXIOM
27 ¬TOWER(t(A,t(B,t(C,t(D,0)))));THM
28 ON(x,v)∧(TOWER(v)∧PARTS(x,v,u))⊃TOWER(u);AXIOM
```

TOWER PROGRAM 2

This tower program makes stronger use of the data structure t(x,y), where x is the top of the tower and y is the rest. It is about 15 times faster than program 1.

```
VAR:   u.x.v.g.z;
PRE_PRED: TOUER.ON,BLOCK;
PREJ3P:   t,A.B,C,D,0;
0N(x_tv)AT0UER(t(v.y))DT0L4ER(t(x.    t(v.y)J);
BLOCK(*):>TOUER(t(x,0>);
ON(A.B);
ON(B.C);
ON(C.O):
ON(D.0>;
BLOCK(A);
BLOCK(B);
BLOCK(C);
BLOCK<D>:

THH:
TOUERIt (A, t 13. t<C. MO.0) ))>);;

CHOICE-STRATEGY-IS:
V|NE/>SUPPOWT ITNn);

EDIT-STRATECY-1S:
DEPTH (GJvLfcNGTH 13) t

ELAPSEO-TIME  -S2S3

NIL  1  7
1  -TOUERtt(D.0»);3  8
2   BLOCK(u)^TOUER(t(u,0));AXIOn
3  -TOUERCHC. t(D.0J));S  8
6  -TOUl-R<t(B. t(C. t(O,0))));7  8
7  -TOUER(t(A,t  (B.t(C.t(0,0)1)I);THH
8  0N(u.x)nT0UERIt (x.v>):>T0UER<t (u. t (x.v)));AXIOM
```