LOGIC PROGRAMMING NEWSLETTER 2

OUTONO DE 1981 / AUTUMN 1981

CONTENTS

EDITOR

Prof. Luís Moniz Pereira
Departamento de Informática
Universidade Nova de Lisboa

PUBLISHED

UNIVERSIDADE NOVA DE LISBOA
Centro de Informática
Faculdade de Ciências e Tecnologia
Quinta do Cabeço — Olivais
1899 Lisboa Codex
Portugal
Tels. 251 25 26 • 251 26 60 • 251 25 55

ACKNOWLEDGEMENT

This publication has been fully subsidized by Junta Nacional de Investigação Científica e Tecnológica (JNICT), Av. D. Carlos I, 126, Lisboa, Portugal.

COLABORATION

The editor thanks António Porto, Eugénio Oliveira, Luís Monteiro and Miguel Filgueiras for their help.

CONTRIBUTIONS

Send to the editor, typed or printed, if possible double spaced, one side of the paper only and a maximum width of 15 cm.

EDITOR'S VIEWPOINT

Logic Programming has been picking up speed in the USA. Within four months two workshops took place in the States. The first, in April, at Syracuse University, NY. The second, in August, aboard the R. M. S. Queen Mary, Long Beach, CA, brought together a motley crew of veteran logic programmers and sympathetic onlookers, some of which may be seen in the photograph. (Yours truly is the one shutting his eyes back at the shutter). It may have not escaped your attention the deliberate metaphor of a transatlantic vessel bringing in logic programming from Europe to the USA.

DESIGN, TYPESETTING AND PRINTING

Serviços Gráficos da Universidade Nova de Lisboa
Av. Miguel Bombarda, 20-1.°
1000 Lisboa

A SOLUTION TO THE MISTER P AND THE MISTER S PROBLEM IN PURE PROLOG

Alain Colmerauer

Groupe d'Intelligence Artificielle
Faculté des Sciences de Luminy
13288 Marseille cedex 2, France

The program is written without any use of "cuts" or "negations by failure".
The arithmetic is done by the predefined predicate:

val(plus(x,y),z)	iff	x+y = z
val(moins(x,y),z)	iff	x−y = z
val(mult(x,y),z)	iff	x*y = z
val(div(x,y),z)	iff	x modulo y = z
val(inf(x,y),1)	iff	x < y
val(inf(x,y,),∅)	iff	not x < y

It runs with a minor change (to avoid a stack overflow) on our Apple II system and takes 11 hours and 39 minutes to find the solution (13,4)!
The system was designed by H. Kanoui, M. Van Caneghem and myself.
It uses a virtual memory on small floppy disks and works even in August without any fan.

```
"Problem of Mister P and Mister S"

Solution(x) -> Pair(x) Property4(p) TrueThat(x,p);

"The four unary properties"

Property1(Not(p)) ->
    Is(p,If(OneKnowsIts(Product),OneKnowsItIsTheOnlyOneSuchThat(ItExists)));

Property2(And(p21,Not(p22))) ->
    Property1(p1)
    Is(p21,If(OneKnowsIts(Sum),OneKnows(p1)))
    Is(p22,If(OneKnowsIts(Sum),OneKnowsItIsTheOnlyOneSuchThat(ItExists)));

Property3(p3) ->
    Property2(p2)
    Is(p3,If(OneKnowsIts(product),OneKnowsItIsTheOnlyOneSuchThat(p2)));

Property4(p4) ->
    Property3(p3)
    Is(p4,If(OneKnowsIts(Sum),OneKnowsItIsTheOnlyOneSuchThat(p3)));

"Some equivalences between unary properties"

Is(p1,If(OneKnowsIts(O),OneKnows(p2))) ->
    Identical(p1,And(p2,AnyOtherPairWithSame(O,p2)));
Is(p1,If(OneKnowsIts(O),OneKnowsItIsTheOnlyOneSuchThat(p2))) ->
    Identical(p1,And(p2,AnyOtherPairWithSame(O,Not(p2))));
Is(if(HasAsA(AlmostProduct,u),p1),p1) ->
    Identical(p1,If(HasAsA(Product,u),p2));
Is(p1,p1) -> Identical(p1,If(HasAsA(Sum,u),p2));

Identical(p,p) -> ;
```

"It is true that a pair x has a given property p"

```
TrueThat(x,ForAnyPairBeyondIt(d,If(p1,p2))) ->
    NextPseudoPair(x',x,d,p1)
    NoPairLeftStartingFrom(x',d,v)
    Or(v,For(x',And(p2,ForAnyPairBeyondIt(d,If(p1,p2)))));
TrueThat(x,And(p1,p2)) -> TrueThat(x,p1) TrueThat(x,p2);
TrueThat(x,If(p1,p2)) -> FalseThatPair(x,p1);
TrueThat(x,If(p1,p2)) -> TrueThatPair(x,p1) TrueThat(x,p2);
TrueThat(x,ItExists) ->;
TrueThat(x,Not(p)) -> FalseThat(x,p);
TrueThat(x,AnyOtherPairWithSame(O,p1)) ->
    TrueThatPair(x,HasAsA(O,u))
    Is(p2,If(HasAsA(O,u),p1))
    TrueThat(x,ForAnyPairBeyondIt(Down,p2))
    TrueThat(x,ForAnyPairBeyondIt(Up,p2));

TrueThatPair(<i,j>,HasAsA(Product,u)) -> val(mult(i,j),u);
TrueThatPair(<i,j>,HasAsA(Sum,u)) -> val(plus(i,j),u);

Or(True,q) ->;
Or(False,For(x,p)) -> TrueThat(x,p);
```

"It is false that a pair x has a given property p"

```
FalseThat(x,ForAnyPairBeyondIt(d,If(p1,p2))) ->
    NextPseudoPair(x',x,d,p1)
    NoPairLeftStartingFrom(x',d,False)
    FalseThat(x',And(p2,ForAnyPairBeyondIt(d,If(p1,p2)))));
FalseThat(x,And(p1,p2)) -> FalseThat(x,p1);
FalseThat(x,And(p1,p2)) -> FalseThat(x,p2);
FalseThat(x,If(p1,p2)) -> TrueThatPair(x,p1) FalseThat(x,p2);
FalseThat(x,Not(p)) -> TrueThat(x,p);
FalseThat(x,AnyOtherPairWithSame(O,p1)) ->
    TrueThatPair(x,HasAsA(O,u))
    Is(p2,If(HasAsA(O,u),p1))
    FalseThat(x,ForAnyPairBeyondIt(d,p2));

FalseThatPair(<i,j>,HasAsA(Product,u)) -> val(eq(mult(i,j),u),Ø);
```

"Domain of the pairs"

```
Pair(<i,j>) -> In(i,<2,99>) In(j, <2,i>);

In(i,<i,j>) ->;
In(i,<i1,j>) -> val(inf(i1,j),1) val(plus(i1,1),i2) In(i,<i2,j>);
```

"Limits of the domain after ordering it"

```
NoPairLeftStartingFrom(<i,j>,Down,False) -> val(inf(i,j),Ø);
NoPairLeftStartingFrom (<i,j>,Up,False) ->
    val(inf(i,1ØØ),1)
    val(inf(1,j),1);
NoPairLeftStartingFrom(<i,j>,Down,True) -> val(inf(i,j),1);
NoPairLeftStartingFrom(<i,j>,Up,True) -> val(inf(i,1ØØ),Ø);
NoPairLeftStartingFrom(<i,j>,Up,True) -> val(inf(1,j),Ø);
```

"From one (pseudo)pair to another"

```
NextPseudoPair(<i',j'>,<i,j>,d,p) ->
    NextInteger(i',i,d)
    TrueThatPseudoPair(<i',j'>,p);

TrueThatPseudoPair(<i,j>,HasAsA(Sum,u)) -> val(moins(u,i),j);
TrueThatPseudoPair(<i,j>,HasAsA(AlmostProduct,u)) -> val(div(u,i),j);

NextInteger(i',i,Down) -> val(moins(i,1),i');
NextInteger(i',i,Up) -> val(plus(i,1),i');;
```

AVL-TREE INSERTION: A BENCHMARK PROGRAMM BIASED TOWARDS PROLOG

Maarten van Emden

University of Waterloo, Canada

I surmise that Prolog compares favourably in terms of efficiency with conventional high-level languages for tasks that do not require arrays or iteration. AVL-tree (height-balanced-tree) insertion is such a task. A Pascal program for it appears in "Program = Algorithm + Data Structure" by N. Wirth.

I was attracted to the idea of comparing a Prolog program with Wirth's for AVL-tree insertion. The most important aspect is readability, but that is hard to compare objectively. Nevertheless, do make the comparison! The Prolog program ran about 3 times faster (using our IBM Prolog) than the Pascal version on "Waterloo Pascal" (an interpreter) and 7 times slower than code compiled by Pascal-VS, an optimizing compiler. This comparison is invalidated by the fact that our IBM Prolog is based on the original Marseille design and therefore quite wasteful of storage.

I therefore look forward to the result of the comparison Ken Bowen is going to make on the DEC-10 using Warren's compiler because there is every reason to believe that its storage management is about as efficient (space-wise) as Pascal's. A listing of my program follows. It is shown the way it was run in Waterloo. On the DEC10 pragmatics may have to be added to achieve the desired level of efficiency.

```
INSERT(NIL,*ELT, AVL(NIL,*ELT,-,NIL),YES).
                                                        /*      ...(5)  */
INSERT(AVL(*LST,*ROOT,*BI,*RST),*ELT,*NT,*HTISCHANGED)
<- .LE(*ELT,*ROOT) & INSERT(*LST,*ELT,*LST1,*LSTISCHANGED)
  & ADJUST(AVL(*LST1,*ROOT,*BI,*RST),*LSTISCHANGED,LEFT,*NT,*HTISCHANGED).
                                                        /*      ...(6)  */
INSERT(AVL(*LST,*ROOT,*BI,*RST),*ELT,*NT,*HTISCHANGED)
<- GE(*ELT,*ROOT) & INSERT(*RST,*ELT,*RST1,*RSTISCHANGED)
  & ADJUST(AVL(*LST,*ROOT,*BI,*RST1),*RSTISCHANGED,RIGHT,*NT,*HTISCHANGED).
                                                        /*      ...(7)  */
ADJUST(*OLDTREE,NO,*,*OLDTREE,NO).                      /*      ...(8)  */

ADJUST(AVL(*LST,*ROOT,*BI,*RST),YES,*LOR,*NT,*HTISCHANGED)
<- TABLE(*BI,*LOR,*BI1,*HTISCHANGED,*TOBEREBALANCED)
  & REBALANCE(AVL(*LST,*ROOT,*BI,*RST,*BI1,*TOBEREBALANCED,*NT).

/*      BALANCE     WHERE       BALANCE     WHOLE TREE    TO BE
          BEFORE    INSERTED      AFTER       INCREASED   REBALANCED    */
TABLE(     -     ,   LEFT    ,     <     ,      YES      ,     NO     ).
TABLE(     -     ,   RIGHT   ,     >     ,      YES      ,     NO     ).
TABLE(     <     ,   LEFT    ,     -     ,       NO      ,     YES    ). /*...(9)*/
TABLE(     <     ,   RIGHT   ,     -     ,       NO      ,     NO     ).
TABLE(     >     ,   LEFT    ,     -     ,       NO      ,     NO     ).
TABLE(     >     ,   RIGHT   ,     -     ,       NO      ,     YES    ).

REBALANCE(AVL(*LST,*ROOT,*BI,*RST)
          ,*BI1,NO,AVL(*LST,*ROOT,*BI1,*RST)).

REBALANCE(*OLDTREE,*,YES,*NEWTREE)
          <- (*OLDTREE => *NEWTREE).

AVL(*ALPHA,*A,>,AVL(*BETA,*B,>,*GAMMA))  =>
AVL(AVL(*ALPHA,*A,-,*BETA),*B,-,*GAMMA).

AVL(AVL(*ALPHA,*A,<,*BETA),*B,<,*GAMMA)  =>
AVL(*ALPHA,*A,-,AVL(*BETA,*B,-,*GAMMA)).

AVL(*ALPHA,*A,>,AVL(AVL(*BETA,*X,*BI1,*GAMMA),*B,<,*DELTA))  =>
AVL(AVL(*ALPHA,*A,*BI2,*BETA),*X,-,AVL(*GAMMA,*B,*BI3,*DELTA))
<- TABLE2(*BI1,*BI2,*BI3).

AVL(AVL(*ALPHA,*A,>,AVL(*BETA,*X,*BI1,*GAMMA)),*B,<,*DELTA)  =>
AVL(AVL(*ALPHA,*A,*BI2,*BETA),*X,-,AVL(*GAMMA,*B,*BI3,*DELTA))
<- TABLE2(*BI1,*BI2,*BI3).

      /*  BI1     BI2     BI3    */
TABLE2(    <   ,   -   ,   >     ).
TABLE2(    >   ,   <   ,   -     ).
```

THE "S-P PROBLEM" REVISITED

David Warren
SRI, USA

David Warren (Depart. of AI, Univ. of Edinburgh, now moving to SRI) has modified the solution to the "S-P problem" by António Porto which appeared in the previous issue of the Newsletter, in an effort to produce a clearer version.

These are David's notes on his program in a letter to António, followed by its listing:

1) The first four statements correspond exactly to the four statements of the dialogue.
2) 'retract' etc. is dispensed with in favour of 'setof', which is really what is needed.
3) My version actualy solves the problem in very much the same way as Max Bramer outlined in AISB-37. It takes approximately 9 seconds. (*)
4) The quantifiers "one", "several", "every" are very close to the treatment of determiners in the natural language question-answering system Chat, which Fernando Pereira and I have implemented.

(*) António Porto comments that his version also runs in the same way and takes approximately 4 seconds, basically because his implementation of the quantifiers is more efficient by not using 'setof' — 'one_and_only_one' fails as soon as a second solution is found, and 'every' also checks that there are several of them.

```
% Mr S and Mr P Problem.
%
% There are two numbers M and N such that 1 < M & N < 100.
% Mr S is told their sum S and Mr P is told their product P. The
% following dialogue takes place:
%
% Statement-1:
%  Mr P: I don't know the numbers.
%        (There are several sum values S that are compatible with
%         the product value P).

statement1(P) :- several(S, compatible(S,P)).

% Statement-2:
%  Mr S: I knew you didn't know them;
%        I don't know them either.
%        (For every product value P that is compatible with the
%         sum value S, statement-1 is true of P; and
%         there are several product values P that are compatible
%         with the sum value S).

Statement2(S) :- every(P, compatible(S,P), statement1(P)),
                 several(P, compatible(S,P)).

% Statement-3:
%  Mr P: Now I know the numbers!
%        (There is just one sum value S compatible with the product
%         value P for which statement-2 is true of S, and that value
%         is S1).

statement3(P,S1) :- one(S,
                        (sumvalue(S), statement2(S),
                         compatible(S,P)),
                    S1).

% Statement-4:
%  Mr S: Now I know them too!
%        (There is just one product value P compatible with the
%         sum value S for which statement-3 is true of P and S,
%         and that value is P1).

statement4(S,P1) :- one(P, (statement3(P,S), compatible(S,P)), P1).

% Question: What are the numbers?
%           (For which sum value S and product value P is state-
%            ment-4 true?)

answer(S,P) :- statement4(S,P).

% [The single solution S = 17, P = 52 is produced in about 9 seconds].

% Definitions of the quantifiers 'one', 'several' and 'every'.

one (X,P,X1) :- setof(X,P,[X1]).

several(X,P) :- setof(X,P,Xs), length(Xs,N), N > 1.

every(X,P,Q) :- \+ (P, \+Q).

% The remaining definitions are compiled:

:-compile(sandp1).

%-----------------------------------
% Supporting definitions for the Mr S and Mr P Problem.

:-public compatible/2, sumvalue/1.

% Sum values range from 4 to 198.

sumvalue(S) :- range(S,4,198).

% The next two clauses are logicaly equivalent to the third clause,
% but are more efficient in the cases that S or P are already known.

compatible(S,P) :- nonvar(S), !,
     Mmax is S/2, S99 is S-99, max(2,S99,Mmin),
     range(M,Mmin,Mmax),
     P is M*(S-M).
compatible(S,P) :- nonvar(P), !,
     sqroot(P,Mmax), P99 is P/99, max(2,P99,Mmin),
     range(M,Mmin,Mmax),
     N is P/M, P is M*N,
     S is M+N.

% Sum value S is compatible with product value P if there are
% numbers M and N in the range 2 to 99 such that S is the sum
% of M and N, and P is the product of M and N. (See above).

compatible(S,P) :-
     range(M,2,99),
     range(N,2,99),
     S is M+N,
     P is M*N.

% Finally, definitions of the predicates 'range', 'max' and 'sqroot'.

range(I,L,M) :- nonvar(I), !, L =< I, I =< M.
range(I,I,_).
range(I,L,M) :- L < M, L1 is L+1, range(I,L1,M).

max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y, !.

sqroot(N,RN) :- N < 181, !, N1 is N*4, N2 is N*2,
        sqroot(N1,RN1,0,N2), RN is RN1/2.
```

```
sqroot(N,RN) :- N < 32768, !, N1 is N*4,
        sqroot(N1,RN1,0,363), RN is RN1/2.
sqroot(N,RN) :- sqroot(N,RN,0,363).

% 'sqroot' expanded to include upper and lower limits
sqroot(N,RN,RN,_) :- N =:= RN*RN, !.
sqroot(N,RN,RN,RN1) :- RN1-RN < 2, !.
sqroot(N,RN,LL,UL) :- ML is (LL+UL+1)/2, M is ML*ML,
        sqrootn(N,RN,LL,UL,ML,M).

% 'sqrootn' sets up for next invocation of sqroot
sqrootn(N,RN,LL,UL,ML,M) :- M > N, !, sqroot(N,RN,LL,ML).
sqrootn(N,RN,LL,UL,ML,M) :- M =< N, sqroot(N,RN,ML,UL).
```

BOOLEAN SATISFIABILITY

Dennis Kibler
University of California, Irvine, USA

The following program solves the quintessential Np-complete problem, that of boolean satisfiability. Such programs seem to require exponential running time. An analysis of the average behavior of the following program would be interesting, but is beyond my means.

To use the program merely type " sat(some boolean expression) ". The program will determine if any assignment of t or f (for true and false) to the variables will make the expression true.

```
%-op(100,fx, '-').
:-op(200,yfx,'&').
:-op(300,yfx,'|').

/* On input, separate & and | from - by a space. */

sat(t):-!.
sat(X|Y):-sat(X);sat(Y).
sat(X&Y):-sat(X),sat(Y).
sat(-X):-not(X).

not(f):-!.
not(X&Y):-not(X);not(Y).
not(X|Y):-not(X),not(Y).
not(-X)-sat(X).
```

A FORWARD CHAINING PROBLEM SOLVER

Paul Morris
University of California, Irvine, USA

This example indicates how an exhaustive forward chaining problem solver may be written in PROLOG. Both breadth-first and depth-first variants are given. The code is contained in three files: PS, PS1 and PS2. Loading PS with PS1 gives the breadth-first variant; PS with PS2 gives depth-first. Two sample domains are given. These are the blocks world and the chimp and bananas world.

The following points are of special interest. States are represented by a list of terms, each term denoting a relationship. Rather than standard STRIPS operators, we have used relational productions à la Stephen Vere. Each term in the left hand side of such a production must match a distinct *relationship in the current state. This requirement precludes the generation of impossible states which would then have to be weeded out. Following Vere, we have factored out the terms common to both sides of the production, and placed them separately. A second point of interest is the use of the list of conditions that are always true. Terms in the left side of productions may match these, rather than items in the current state. Terms in the right side matching these are never added to the state. This allows us to represent the blocks world, including movements to and from the table, with but a single operator. Movements from the table to the table are avoided by a feature of the program which prevents duplication of previously reached states.*

This program was first written in UCI LISP and then rewritten in DEC-10 PROLOG. The compiled PROLOG version runs more than four times faster than the compiled LISP version.

The file PS:

```
/* Forward-chaining problem solver */
/* PS+PS1 = breadth-first */
/* PS+PS2 = depth-first */

:-op(500, xfy, '&').
:-op(500, xfy, '=>').

:-public solve/1, access/3.

solve(Goals):- access(S,N,P), included(Goals,S), n1, write(P).

/* access, on backtracking, does a complete traversal of the state
space, without repetition. S is bound to each state, while P gives the
path (operator sequence) needed to reach it. N is the path length. */

/* access is defined in PS1 and (differently) in PS2 */

applic(Opr,S1,S) :-
      action(Com, Pre => Post, Opr),
      always(Als),
        deletconds(S1,Als,Pre,S2,Al1),
        deletconds(S2,Al1,Com,S3,Al2),
        addconds(S3,Com,S4),
        addconds(S4,Post,S).

deletconds(S,Als,[],S,Als).
deletconds(S,Als,[C, . .Z],S2,Al2) :-
      deletconds(C,S,Als,S1,Al1),deletconds(S1,Al1,Z,S2,Al2).

addconds(S,[],S).
addconds(S,[X, . .Y],S2) :- addcond(X,S,S1), addconds(S1,Y,S2).

addcond(X,S,S) :— always(Y),member(X,Y),!.
addcond(X,S,[X, . .S]).

deletcond(C,S,Als,S1,Als) :— delete(C,S,S1).
deletcond(C,S,Als,S,Al1) :— delete(C,Als,Al1).

delete(X,[X, . .Y],Y).
delete(X,[Y, . .Z],[Y, . .W] :— delete(X,Z,W).

included(G1&G2,L) :— member(G1,L),included(G2,L),!.
included(G,L) :— member(G,L).

member(X,[X, . .Y]).
member(X,[Y, . .Z]) :— member(X,Z).
```

The file PS1:

```
/* Must also load PS */

access(State,N,Path):—
      abolish(reached,3),
```

```
        reach(State,N,Path),
        write(N), ttyflush, /* to see what's happening */
        assertz(reached(State,N,Path)).

reach(SS,0,start):— initstate(S),sort(S,SS).
reach(SS,N,Path+Opr):—
            reached(S1,N1,Path),
        N is N1+1,
        applic(Opr,S1,S),
          sort(S,SS),
        \+(reached(SS,_,_)).
```

The file PS2:

```
/* Must also load PS */

access(State,N,Path):—
        abolish(seenstate,1),
        initstate(S0), sort(S0,SS0),
        reachable (SS0,0,start,State,N,Path),
        write(N),ttyflush,        /* to see what's happening */
        assert(seenstate(State)).

reachable(S,N,P,S,N,P).
reachable(S,N,P,Sf,Nf,Pf):—
        N1 is N+1,
        applic(Opr,S,S1),
        sort(S1,SS1),
        \+(seenstate(SS1)),
        reachable(SS1,N1,P+Opr,Sf,Nf,Pf).
/* Notice the six variables in reachable. The extra three are needed to
transport the answers back from the bottom of the loop. We could get
by without them by inverting the loop, but then backtracking would
occur, undesirably, at the other end. */
```

The blocks world:

```
action( [clear(X)],
         [on(X,Y),clear(Z)] => [on(X,Z),clear(Y)],
         move(X,Z) ).

always([clear(table)]).

initstate( [on(a,b),on(c,table),on(b,table),
            clear(a),clear(c)] ).
```

The chimp and bananas world:

```
always([pos(a),pos(b),pos(c)]).

action([pos(X),pos(Y),low(chimp)],
         [at(chimp,X) ]=> [at(chimp,Y)],
         gofrom(X,Y) ).

action([pos(X),pos(Y),low(chimp)],
       [at(chimp,X),at(box,X)] => [at(chimp,Y),at(box,Y)],
       pushbox(X,Y)).

action([pos(X),at(box,X),at(chimp,X)],
       [low(chimp)] => [high(chimp)],
       climbox(X)).

action([high(chimp),pos(X),at(chimp,X),at(bananas,X)],
       [nobananas] => [hold(bananas)],
       grabananas).

action([pos(X),at(box,X),at(chimp,X)],
       [high(chimp)] => [low(chimp)],
       climbdown).

inistate([at(chimp,a),at(box,b),at(bananas,c),
    low(chimp),nobananas]).
```

A SHORT CUT TO MORE INFORMATIVE ANSWERS

Veronica Dahl
Buenos Aires University, Argentina

Most Prolog natural language (NL) data base systems implemented to date, base retrieval upon the evaluation of a formula obtained by parsing a user's NL query.

Typically, language coverage is far wider for input than for output, given that much more attention has been dedicated to sentence analysis than to generation. Output sentences are usually laconic answers to yes-no, who or how much/many questions, constructed around a few key words and phrases.

This is quite appropriate for straightforward, naïve answers, but hardly mimicks ordinary human communication, in which the questioner's assumptions are detected, taken into consideration and — assuming no stonewalling intentions — corrected and supplemented when felt necessary.

Some systems — cf. those implemented for Spanish, French, Portuguese and English from [Dahl 1977] — partially capture this ability by detecting certain failed presuppositions (namely, those induced by NL quantifiers or by certain kinds of plural), with the aid of a third logical value (e.g. "pointless"). Because of the lack of generating capabilities, however, failed presuppositions are detected, but neither identified nor corrected. This would involve restating at least the presupposition's internal representation into NL terms.

The shortest short cut to avoid developing an output grammar might be to store both representations (i.e., the NL and the formal one) of each presupposition, as a side effect of parsing. This, however, — besides too much nearing the unaesthetical brute force approach — might mean redundant work in some cases (e.g. when for some reason the system decides to overlook a given failure), and may lose useful information that is explicity only in the internal formula (e.g., type constraints).

I have recently implemented another Prolog short cut which postpones decisions on when and what to output until all the information on the input sentence becomes available (i.e., at the evaluation stage). Although also based upon some fixed NL expressions, it acquires a fluent-like touch by incorporating the names of relations and items named in the query or retrieved, plus the additional information it can deduce from the question (e.g. semantic types, implicit meanings of quantifiers, etc.).

Thus, an input sentence as:

> *"The salesman that lives in Temperley earns 4 million",*

which used to get the fixed answer:

> *"There is some false presupposition in your statement"*

now gets the answer:

> *"You are wrongly assuming that there is exactly one human that is a salesman and that lives in Temperley".*

In order for the system to make subtler decisions and provide richer answer, extra non-traditional truth values may be added. This is quite inexpensive in the kind of systems we are concerned with. For instance, different values can be distinguished for a quantifier's failed presupposition and for distributive plurals with mixed truth values (as in "Al teachers like Philosophy" with respect to a world where some do while others don't).

The system is then informed about the specific type of presupposition that has failed, and this information can be easily carried on — always via truth values — to higher levels of a sentence.

This approach has the two following advantages:

- *— Because multiple values are always handy for an overall view by the system, decisions can either be local to the subformula evaluated, or be taken after an analysis of the particular presuppositions in a given query (for instance, a series of embedded false presuppositions might not be all reported; different kinds of presuppositions might interact, etc). This seems a more human-like behaviour than a standard, rigid treatment of all presuppositions, irregardless of context.*
- *— Since fixed expressions, besides formula-extracted basic information, are the basis of this approach, the extra power is gained while maintaining transposal to various natural languages quite a simple matter.*

A more comprehensive approach, of course, would be to output informative and linguistically rich messages through a generating grammar. Ideally, the same one should serve both for parsing and generation.

This, however, seems a difficult matter for the present state of the art in Prolog grammars, and might require more evolved forms of MGs [Colmerauer 1978]. In the meantime, the solution outlined here might prove a good compromise yielding immediate, non-expensive results.

ACKNOWLEDGEMENT

Thanks are due to Lisbon's Laboratório Nacional de Engenharia Civil, for having provided the computer facilities that lack at the Buenos Aires University.

REFERENCES

COLMERAUER, A. (1978), Metamorphosis grammars. In: Natural language communication with computers, vol. I, Springer-Verlag.

DAHL, V. (1977), Un système déductif d'interrogation de banque de données en espagnol. Thèse de Troisième Cycle, Univ. d'Aix-Marseille, France.

T-PROLOG: A VERY HIGH LEVEL SIMULATION SYSTEM

I. Futó, J. Szeredi

Inst. for Coordination of Computer Techniques / SZKI /
H-1368 Budapest, P.O.B. 224, Hungary

For modeling we found useful to combine the time concept of discrete simulation languages and non-procedural programming concepts of PROLOG. The result is a discrete simulation system called by us T-PROLOG [2].

The basic elements of the system are:

1. *The notion of process /in the sense of [3]/.*
   There can be several processes in the system running conceptually in parallel. A PROLOG-like goal corresponds to each of them. Each process is executed according to the control mechanism of PROLOG, influenced by inter-process communication.
2. *The notion of resource.*
   Resources are model elements that can be used only by one process at a time. The use of a resource can take time and may be returned or deleted after use. Resources also may be created at program initialisation or dynamically during program execution.
3. *The notion of internal time.*
   A duration time can be assigned to certain program elements independently of their real execution time. An internal clock is maintained by the system which is used for scheduling the processes.
4. *Messages and conditions.*
   Processes can wait for fulfilment of conditions or messages. Conditions can be formulated using the full power of PROLOG.
5. *Backtrack.*
   The whole system including the scheduler can backtrack into a previous state and try an alternative path of control.
   Backtrack is caused either by a failure in the execution of a process or by a deadlock or when the prescribed termination time of a process is passed.
   Choice points are implicitly generated at resource allocation to try all meaningful distributions of resources among processes.

To illustrate programming in T-PROLOG we give a definition of the five philosophers problem.

```
Resource(Fork(A)).
Resource(Fork(B)).
Resouce(Fork(C)).
Resource(Fork(D)).
Resource(Fork(E)).
Satiated(n):
        Good_fork(n,f1,f2), Seize(f1), Seize(f2),
        Eat, Release(f1), Release(f2).
Eat :.
    During(10).
Good_fork(1,Fork(A),Fork(B)).
Good_fork(2,Fork(B),Fork(C)).
Good_fork(3,Fork(C),Fork(D)).
Good_fork(4,Fork(D),Fork(E)).
Good_fork(5,Fork(E),Fork(A)).

New(Satiated(1),Phil1,0,30), New(Satiated(2),Phil2,0,30),
New(Satiated(3),Phil3,0,30), New(Satiated(4),Phil4,0,30),
New(Satiated(5)Phil5,0,30).
```

Running the goal (given at the end of the example) causes various deadlock situations to occur, after which the system is able to find all possible solutions.

The built-in procedures of T-PROLOG used in the example are:

- *— SEIZE and RELEASE for seizing and releasing resources,*

— *DURING for suspending a process unconditionally for a given time duration,*
— *NEW for creating a process with a given goal, identifier, start time and termination time.*

The current implementation of T-PROLOG is based on the MPROLOG system [1], and uses special built-in procedures for handling several pseudo parallel threads of control.

REFERENCES

[1] J. BENDL, P. KÖVES, P. SZEREDI: The MPROLOG system. Preprints of Logic Programming Workshop, Debrecen (Hungary), 1980, pp. 201-210.

[2] I. FUTÓ, J. SZEREDI: T-PROLOG: general information manual. Institute for Co-ordination of Computer Techniques, Budapest, 1980.

[3] R. E. NANCE, V. TECH: The time and state relationship in simulation modelling. Comm. ACM. Vol. 24. No. 4. Apr. 1981, pp. 173-180.

ALL SOLUTIONS

Luís Moniz Pereira
António Porto

Departamento de Informática
Universidade Nova de Lisboa
1899 Lisboa, Portugal

Any Prolog programmer sooner or later feels the need for a predicate capable of producing the set of all solutions to a given problem.

Those not fortunate enough to have a Prolog system offering such a predicate as a built-in feature usually resort to ad-hoc techniques for achieving its effect in a particular setting. We show a compact, reasonably efficient and sound implementation of such a predicate, that anybody can use since it is written in Prolog itself.

The predicate is

all (T , G , L)

and it reads " all instances of the term T *for which the goal* G *is satisfied are the members of list* L *". L is required to be non-empty, so* 'all' *fails if* G *has no solution.*

The term T *is just a template for building* L*, so free variables within will not be bound upon execution of* 'all'.

G *can be any valid goal expression in Prolog, including* 'cut's *(which only affect the evaluation of* G *within the evaluation of* 'all'*) and* 'all's *(whose nesting is very useful for structuring sets of solutions). Furthermore,* G *can be of the special form*

G1 same T1

where G1 *is any goal expression and* T1 *is any term. This variant allows the distinction between two roles of the free variables appearing in* G *but not in* T*:*

If G *is not of the* 'same' *type, the different solutions of* G *for which instances of* T *are put in* L *can correspond to different instantiations of any free variable in* G*, and* 'all' *acts as a deterministic predicate.*

If, however, G *is of the form* 'G1 same T1'*, the different solutions of* G1 *for which instances of* T *are put in* L *must correspond to the same instance of* T1*, which remains enforced within the execution of* 'all'*; backtracking into* 'all' will produce another instance of T1 *and another corresponding list* L*, until no more solutions to* G1 *exist and 'all' finally fails.*

All this is best seen with an example. Suppose we have the following micro data base:

```
drinks(john,tea,hot).
drinks(john,milk,hot).
drinks(john,milk,cold).
drinks(john,milk,warm).
drinks(john,beer,cold).
drinks(john,wine,cold).

drinks(bill,milk,cold).
drinks(bill,beer,cold).
drinks(bill,beer,warm).

drinks(joe,beer,cold).
drinks(joe,wine,cold).
drinks(joe,wine,warm).
drinks(joe,tea,hot).
drinks(joe,tea,warm).
drinks(joe,tea,cold).
```

Some natural language questions follow, along with the corresponding formulation in terms of the 'all' *predicate, and its solution(s).*

Who drinks?

```
all( P, drinks(P,_,_), X).

X = [ john, bill, joe].
```

Who drinks the same drink?

```
all( P, drinks(P,D,_) same D, X).

X = [ john, joe] , D = tea   ;
X = [ john, bill] , D = milk   ;
...
```

Who drinks each drink?

```
all( D-Ps, all( P, drinks(P,D,_) same D, Ps), X).

X = [ tea-[john,joe], milk-[john,bill], ... ].
```

Who drinks (and at which temperatures) each drink?

```
all( D-PT, all( P:Ts, all( T, drinks(P,D,T) same (D,P), Ts) same D, PT), X).

X = [ tea-[ john:[hot], joe:[hot,warm,cold]], milk-[ john:[hot,...],...],...].
```

The Prolog definition of 'all' *now follows:*

```
?- op(50,xfx,same).

all(T,G same X,S) :- !, all(T same X,G,Sx), produce(Sx,S,X).

all(T,G,S) :- asserta(one(end)), solve(G), asserta(one(T)), fail.

all(T,G,S) :- set(S).

solve(G) :- G.

set(S) :- build(S,[]), ( S=[], !, fail ;
                          asserta(set(S)), fail ).
```

short communications

```
set(S) :- retract(set(S)).

build(NS,S) :- retract(one(X)), ( nonvar(X), X=end, NS=S ;
                                  join(S,X,XS), build(NS,XS) ), !.

join(S,X,S) :- in(S,X).

join(S,X,[X|S]).

in([X|_],X).

in([_|S],X) :- in(S,X).

produce([T1 same X1|Tn],S,X) :- split(Tn,T1,X1,S1,S2),
                                ( S=[T1|S1], X=X1 ;
                                  !, produce(S2,S,X) ).

split([],_,_,[],[]).

split([T same X|Tn],T,X,S1,S2) :- split(Tn,T,X,S1,S2).

split([T1 same X|Tn],T,X,[T1|S1],S2) :- split(Tn,T,X,S1,S2).

split([T1|Tn],T,X,S1,[T1|S2]) :- split(Tn,T,X,S1,S2).
```

Some remarks should be made:

1) The non-logical predicates 'asserta' *and* 'retract' *are called from* 'all' *and* 'build' *just to implement a stack where solutions are kept during backtracking within* G.

2) The predicate 'get' *is defined so as to recover the space used by the recursive execution of* 'build', *instead of calling* 'build' *directly from* 'all'.

3) 'solve' *is necessary, so that any* 'cut's *within* G *do not affect the clauses for* 'all'.

4) There is some time lost in keeping L *free of repeated elements. For applications where this feature is not necessary one can define a faster* 'all' *by changing the clauses for* 'build', 'produce' *and* 'split' *as follows:*

```
build(NS,S) :- retract(one(X), ( nonvar(X), X=end, NS=S ;
                                 build(NS,[X|S]) ), !.

produce([T1 same X1|Tn],S,X) :- split(Tn,X1,S1,S2),
                                ( S=[T1|S1], X=X1 ;
                                  !, produce(S2,S,X) ).

split([],_,[],[]).

split([T1 same X|Tn],X,[T1|S1],S2) :- split(Tn,X,S1,S2).

split([T1|Tn],X,S1,[T1|S2]) :- split(Tn,X,S1,S2).
```

5) Where, using DECsystem-10 Prolog's predicate 'setof', *one would write*

setof(X, p(X,Y), S) *and* setof(X, Y^p(X,Y), S) ,

we would write, respectively,

all(X, p(X,Y) same Y, S) *and* all(X, p(X,Y), S) ,

with the difference that we do not sort S.

For natural language processing we prefer our version, since hidden variables do not have to be existentially quantified explicitly.

6) This 'all' *has been tested and used extensively.*

community news & events

FIRST INTERNATIONAL
LOGIC PROGRAMMING CONFERENCE

September 1982
Marseille, France

CALL FOR PAPERS

The intention of this conference is to identify and encourage research into the theory, implementation and applications of logic as a programming language. Papers are sought in the following areas:

1) The use of logic as a *computational formalism* for program development, database description and query, natural language processing, knowledge based systems, and other applications.
2) The *design and implementation* of logic programming systems, both on conventional von Neumann machines and on parallel computer architectures; improved control methods.
3) *Language extensions* such as the use of metalanguage and set theoretic data structures and operations.

Submission of papers:

Papers should be written in English, include an abstract (of approximately 100 words) and not exceed 20 double-spaced pages (5000 words). Send four copies of each paper to the programme committee chairman by March 1, 1982:

Alain Colmerauer

Groupe d'Intelligence Artificielle
Université d'Aix-Marseille II
70 route Léon Lachamps, Case 901
13288 Marseille Cedex 2
FRANCE

Authors will be notified by May 15. Final version due June 15.

Conference location:

The conference will take place at the

Luminy campus of the University of Aix-Marseille II, on the southern outskirts of Marseille, probably on the second week (6-10) of September.

Programme Committee:

K. Bowen, Syracuse, USA
M. Bruynooghe, Leuven, Belgium
K. Clark, London, England
A. Colmerauer, Marseille, France
M. van Emden, Waterloo, Canada
H. Gallaire, Marcoussis, France
R. Kowalski, London, England
L. M. Pereira, Lisbon, Portugal
A. Robinson, Syracuse, USA
P. Roussel, Marseille, France
P. Szeredi, Budapest, Hungary
S. Tarnlund, Uppsala, Sweden

MPROLOG

A MODULAR
LOGIC PROGRAMMING LANGUAGE
FOR REAL LIFE APPLICATIONS

Szki

What is PROLOG?

PROLOG is a very high level programming language based on mathematical logic. It represents a really new direction in programming by supporting both descriptive, non-algorithmic and imperative, procedural approaches. PROLOG is especially useful for solving *complex* problems with relatively small human effort. The language is very simple and *human-oriented*; it can be learnt in a very short time by people without previous computing practice.

The main application areas of PROLOG are

— computer aided design (e.g. pharmacology, architecture)
— artificial intelligence
— symbol manipulation
— question answer systems
— very high level simulation (T—PROLOG).

What is MPROLOG?

MPROLOG is an efficient, modular PROLOG system supporting both the development phase of programming and efficiency tuning for production programs.

Its components are:

— the *pretranslator,* for converting MPROLOG modules into an efficiently executable internal form
— the *consolidator,* for linking separately pretranslated modules into a program
— the *interpreter,* for interpretation of a consolidated program
— the *Program Development Subsystem* (PDSS).

The main component of the system is the MPROLOG interpreter. It applies advanced implementation techniques which result in a high speed of execution and relatively low storage requirements. There are about 200 built-in procedures incorporated in the interpreter; they provide e.g. arithmetic, input--output, data base handling and search space control functions. There is a convenient interface for including new user-defined "built-in" procedures for tuning purposes.

The component of the system the user first meets is the Program Development Subsystem. PDSS provides a user-friendly environment for interactive development of MPROLOG modules. It contains a *dedicated editor,* supplies various *testing aids,* also supports tuning by providing facilities for *program measurement.* PDSS itself is written in MPROLOG and runs under control of the interpreter.

Technical conditions

The MPROLOG system is implemented on assembly level using CDL2 (Compiler Development Language) as a production tool. The system is available at present on SIEMENS 7.755, IBM 3031 and VAX—11/780. The minimal storage requirement is 512 Kbyte. The installation of the system in a new computer environment takes 2—6 months depending on availability of CDL2 for the given architecture.

Szki
INSTITUTE FOR CO-ORDINATION
OF COMPUTER TECHNIQUES HUNGARY
1054—BUDAPEST, Akadémia u. 17.
Tel.: 129—600
Telex: 22—5381

A PROPOSAL

Maarten van Emden

There is widespread confusion about the role of the "slash". I agree with Warren, who maintains it is just a control message to the interpreter which does not affect the logic of any program in which it appears.

To avoid confusion I propose to use two varieties of slash, say, green and red. To the interpreter they mean the same. When a green slash is removed from a clause, and no slashes remain in the partition for that clause, the logical reading of the partition is intended to be true of the relation the programmer wants computed. When a red slash is removed, this is not necessarily the case.

An example of a green slash:

$$\max(x,y,y) \leftarrow x \leq y \;\&/$$
$$\max(x,y,x) \leftarrow x \geq y$$

An example of a red slash:

$$\max(x,y,y) \leftarrow x \leq y \;\&/$$
$$\max(x,y,x)$$

To say that a slash "changes the meaning" of a Prolog program only adds to the confusion. Slashes do no such thing, not even red ones. Only programs without any slash are within the language of logic and, hence, only these have meaning.

A PROLOG INTERPRETER

Grant Roberts's Prolog interpreter for IBM 370, 3031, 4341, and similar machines is now finally being distributed by the University of Waterloo. Note that this interpreter runs only under the VM/CMS operating system. A license for educational institutions costs Can $500 a year. Address enquiries to Sandra Ward, Department of Computing Services, University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada.

NEW *PROLOG* DEVELOPMENTS IN HUNGARY

Péter Szeredi

Institute for Co-ordination of Computer Techniques / SzKI /
H-1368 Budapest, P.O.B. 224, Hungary

The work on implementation and application of PROLOG in Hungary till middle 1980 was presented on the Logic Programming Workshop in Debrecen in [Bendl 89] and [Sántáné 80].

Some interesting new *PROLOG applications* started after that time are the following:

1. Dataflow modelling ([Domán 81])
   An implementation of PARAFLOG, a higher level applicative dataflow language has been developed using PROLOG. It consists of two programs: the *translator* converts PARAFLOG programs into a /universal/ dataflow graph which is then interpreted by the *dataflow simulator.*

2. Documentation ([Fidrich 80])
   The program helps in production of program documentation or other textual objects according to some standards.
   It provides means for handling various forms of requirements /e.g. on the form or contents of text/ and for stepwise refinement of the structure of a document in accordance with the requirements.

3. CAD in mechanical engeneering
   ([Márkus 81], [Molnár 81])
   PROLOG is used in the area of production control for the following purposes:
   - — modelling machine parts with the aim of helping in computer aided classification of machine parts;
   - — supporting design of production control systems: scheduling a shop-floor level production control of an integrated manufacturing system;
   - — designing fixtures from a bounded set of elements.

4. CAD in electronic engeneering
   ([Pásztorné 81])
   Two programs are being developed:
   - — supporting the design and checking of printed circuit boards;
   - — synthesis of circuits for given Boolean functions.

There is also much work is progress on the new Hungarian PROLOG implementation, the MPROLOG system (see [Bendl 80] and for [Köves 81] an overview). After the first phase of tuning the *interpreter* turned to be slightly faster than our old PROLOG, but the process of tuning is still continuing. A lot of built-in procedures have been introduced, e.g. for formatted output of terms, breakpoint handling etc. The *Program Development Subsystem* has been extended with means for program measurement to allow tuning of PROLOG programs. New versions of the other two system components, the *pre-translator* and *consolidator* have also beeen produced on the basis of experience of one year in use. Preparations are being made to include an *MPROLOG compiler* into the system planned for 1982. The whole system has been recently ported to the VAX 11 computer and runs under VMS.

Most of new application projects are now being realized in MPROLOG, furthermore there is an activity to convert "old-PROLOG" programs to MPROLOG in order to increase their efficiency. Means have been developed to support this conversion process.

Two of the converted programs, a drug design application and the T-PROLOG interpreter (see short communications) underwent also a tuning phase with very promising resulte [Szeredi 81].

REFERENCES

[Bendl 80]
J. Bendl, P. Köves, P. Szeredi: The MPROLOG system. Preprints of the Logic Programming Workshop (ed. S-Å. Tärnlund), Debrecen, Hungary, 1980, pp. 201--209.

[Domán 81]
András Domán: An applicative language for highly parallel programming, SzKI Technical Report, Budapest, 1981.

[Fidrich 80]
Ilona Fidrich: User's guide to program documentation system DOKSI (Hungarian) SzKI, Technical Report, Budapest, 1980.

[Köves 81]
Péter Köves, Péter Szeredi: A programming support environment for PROLOG program development. Paper presented on the Workshop "Logic Programming for Intelligent Systems".
Aug 1981, Los Angeles, California, USA.

[Márkus 81]
A. Márkus, B. E. Molnár, E. Szelke: Logic programming in the design of production control systems. To appear in the Proceedings of Compcontrol '81, Nov 1981, Varna, Bulgaria.

[Molnár 81]
B. E. Molnár, A. Márkus: Logic programming in the modelling of machine parts. To appear in the Proceedings of Compcontrol '81, Nov 1981, Varna, Bulgaria.

[Pásztorné 81]
Katalin Pásztorné-Varga: A solution of a CAD problem in PROLOG. Paper presented on the Workshop "Logic Programming for Intelligent Systems"
Aug 1981, Los Angeles, California, USA.

[Sántáné 80]
E. Sántáné-Tóth, P. Szeredi: PROLOG applications in Hungary. Preprints of the Logic Programming Workshop (ed. S-Å. Tärnlund), Debrecen, Hungary, 1980, pp. 177-189.

[Szeredi 81]
Péter Szeredi: Mixed language programming- a method for producing efficient PROLOG programs.
Paper presented on the Workshop "Logic programming for Intelligent Systems"
Aug 1981, Los Angeles, California, USA.

INDUSTRIAL REALIZATION IN PROLOG

Ch. Giraud, J. F. Pique and P. Sabatier

In one month, realized for the french company Cap Sogeti a natural language consultable database. Written in Prolog, this database system describes a french public administration consisting in different types of dependent organisms with their staffs (names, functions, ranks and titles), addresses and telephone numbers.

CONSULTANCY SERVICE IN EXPERT SYSTEMS AND PROLOG

Expert Systems Ltd has formed a pool of associated consultants who are actively involved in work on expert systems, Prolog, and related areas such as knowledge aquisition. These are all qualified, experienced people who could help your organisation take its first step in the field of knowledge engineering.

Specialist experience in any subject is nearly always in short supply even more so in a new subject in which interest is growing rapidly.

If you wish to take advantage of our consultancy service, please contact us directly and we will do our best to accomodate you.

Alternatively, if you have practical experience in the techniques of expert systems, knowledge engineering, knowledge acquisition or programming in Prolog, we would be pleased to include you in our pool of associated consultants.

Over the next year we expect an increasing number of companies will require consultants to advise them on where and how expert systems and related techniques would be of benefit to them. Expert Systems Ltd aims to fulfil this need through its associated consultants.

If you wish to add your particular expertise to the pool of resources already available through Expert Systems Ltd, please complete the form below and return this sheet in the enclosed envelope. Please also include a single-sided sheet containing details of qualifications and a brief curriculum vitae, including availability, percentage of time available, and approximate consultancy rates required.

Write to:

Knowledge Engineers & Designers
of Expert Systems
34 Alexandra Road, Oxford OX2 0DB
Telephone (0865) 42206

published logic programming references

To help make published work by the Logic Programming Community more easily accessible, references to that work should be made to papers published in journals or conference proceedings where possible, rather than to internal reports. To this end, a first list is printed here that will be successively updated. Please help by sending references to your own work.

BELOVARI, G. & CAMPBELL, J. A.
— "Generating contours of integration: an application of PROLOG in simbolic computing"
5th Conference on Automated Deduction, Les Arcs
Lecture Notes in Computer Science, vol. 87
Springer Verlag, 1980.

BOWEN, K. A.
— "Prolog"
ACM 1979 Annual Conference, Detroit M. I., USA.

BRUYNOOGHE, M.
— "Analysis of dependencies to improve the behavion of Logic programs"
5th Conference on Automated Deduction, Les Arcs
Lecture Notes in Computer Science, Vol 87
Springer Verlag, 1980
— "Intelligent backtracking for an interpreter of Horn Clause logic programs"
Proceedings, Colloquia Mathematica Societatis Janos Bolyai, 26
Mathematical Logic in Computer Science
Salgótarjan (Hungary) 1978
North-Holland, 1981
— "Solving combinatorial search problems by intelligent backtracking"
Information Processing letters, no. 1, 1981

BUNDY, A. & BIRD & LUGER & MELLISH & PALMER
— "Solving mechanic problems using meta-level inference"
Proceedings IJCAI-6, Tokyo, 1979

BUNDY, A. & WELHAM, B.
— "Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation"
5th Conference on Automated Deduction, Les Arcs
Lecture Notes in Computer Science, vol. 87
Springer-Verlag, 1980

CLARK, C. L. & McCABE, F. G.
— "IC PROLOG: Language features"
Proceedings of the Logic Programming Workshop
Debrecen, Hungary, 1980

CLARK, C. L. & TÄRNLUND, S.-Å.
— "A first order theory of data and programs"
IFIP '77
North-Holand, 1977

COLMERAUER, A.
— "Metamorphosis Grammars"
Natural Language Communication with Computers
Lecture Notes in Computer Science, Vol. 63
Springer-Verlag, 1978
— "Sur les bases théoriques de PROLOG"
AFCET Groplan Bulletin, no. 9, 1979.

DARVAS, F. & FUTO, I. & SZEREDI, P.
— "Logic based program system for predicting drug interactions"
Int. J. Bio-Medical Computing, no. 9, 1978

DELIYANI, A. & KOWALSKY, R.
— "Logic and semantic networks"
CACM, vol. 22, no. 3

DINCBAS, M.
— "A Knowledge-based expert system for automatic analysis and synthesis in CAD"
Proc. IFIP Congress '80
Tokyo, 1980.

FUTO, I. & DARVAS, F. & CHOLNOKY, E.
— "Pratical application of an AI language"
Proc. of the Hungarian Conference on Computing
Budapest, 1977

FUTO, I. & SZEREDI, P.
— "Some implemented and planned PROLOG applications"
Logic and Data Bases, ed. by Gallaire and Minker
Plenum Press, 1978

GALAIRE, H. & LASSERRE, C.
— "Controlling knowledge deduction in a declarative approach"
Proc. IJCAI-6, Tokyo, 1979

HOGGER, C. J.
— "Goal-oriented derivation of logic programs"
Mathematical Foundations of Computer Science 1978
Proceedings, 7th Symposium Zakopane, Poland 78
Lectures Notes in Computer Science, Vol. 64
Springer-Verlag, 1978

KOWALSKY, R.
— "Predicate Logic as a Programming Language"
Proc. IFIP Congress '74
North-Holland, 1974
— "Algorithm=Logic+Control"
CACM, vol. 22, no. 7, 1979
— Logic for problem solving
Artificial Intelligence Series
The Computer Science Library
North-Holland, 1979

MARKUSZ, Z.
— "How to design variants of flats using the programming language PROLOG based on mathematical logic"
Proc. IFIP 1977
North-Holland, 1977

McDERMOTT, D.
— "The PROLOG phenomenon"
Sigart Newsletter, July 1980

MONTEIRO, L.
— "An extension to Horn clause logic allowing the definition of concurrent processes"
International Colloquium on Formalization of Programming concepts, Peniscola
Lecture Notes in Computer Science, Vol. 108
Springer-Verlag, 1981

PEREIRA, E. & WARREN, D.
— "Definite clause grammars for language analysis — A survey of the formalism and a comparison with augmented transition networks"
Artificial Intelligence Journal, vol. 13, no. 3, 1980

published logic programming references

PEREIRA, L. M. & PORTO, A.
— "Selective Backtracking for Logic Programs"
5th Conference on Automated Deduction, Les Arcs
Lecture Notes in Computer Science, vol. 87
Springer-Verlag, 1980

PEREIRA, L. M. & MONTEIRO, L. F.
— "The semantics of parallelism and co-routining in Logic Programming"
Proceedings, Colloquia Mathematica Societatis János Bolyai, 26
Mathematical Logic in Computer Science
Salgótarjan (Hungary) 1978
North-Holland, 1981

ROBINSON, J. A.
— Logic: form and function
Edinburgh University Press, 1979

TÄRNLUND, S.-Å.
— "An interpreter for the programming language predicate logic"
Proc. IJCAI, Tiblisi, 1975
— "Horn clause computability"
BIT, vol. 17, no. 2, 1977

van EMDEN
— "Programming in resolution logic"
Machine Intelligence, no. 8, 1977

van EMDEN,. M. H. & KOWALSKY, R.
— "The semantics of predicate logic as a programming language"
JACM, vol. 23, no. 4, 1976

WARREN, D. H. & PEREIRA, L. M. & PEREIRA, F.
— "Prolog — the language and its implementation compared with LISP"
Sigplan Notices, vol. 12, no. 8 and Sigart Newsletter, no. 64, August 1977.

abstracts

from Ames, Iowa:

The Recursion-Theoretic Complexity of the Semantics of Predicate Logic as a Programming Language

Howard A. Blair

Each set of definite clauses has associated with it an operator T mapping Herbrand structures to Herbrand structures. The set of definite clauses P is regarded as an abbreviation of its "if and only if" version, CDB(P). The fixed points of T are the Herbrand models of CDB(P). The ascending ($\uparrow$) and descending ($\downarrow$) ordinal powers of T are defined. The descending ordinal powers of T reach a fixed point that is the maximal Herbrand model of CDB(P). The set -A of formulas provable by the negation-as-failure rule corresponds to $IN - T\downarrow^{\omega}$. However, the maximal Herbrand model corresponds to $IN - T\downarrow^{\lambda}$ for λ usually larger than ω. Thus the negation-as-failure rule is often incomplete for various sets of definite clauses.

Two main results are reported here: 1) For some sets of definite clauses P, $\lambda = \omega_1$, (λ is always $\leq \omega_1$) where ω_1 is the least non-constructible ordinal. 2) For some sets of definite clauses P, $IN - T\downarrow^{\lambda}$ is TT^1_1–complete, from which it follows that the problem of deciding Herbrand validity is TT^1_1–complete.

from Budapest:

A Very High Level Discrete Simulation System T-PROLOG

I. Futó
J. Szeredi

T-PROLOG a very high level simulation language is presented. It has the following properties.

— the system takes over part of the problem solving effort from the user.
— a built-in backtrack mechanism permits backtracking in time in case of a deadlock.
— it changes automatically and dynamically the simulation model on the basis of logical consequences.
— a more advanced process communication mechanism is presented for the user.

The processes are synchronized by a built-in scheduler.

Mixed language programming — a method for producing efficient PROLOG programs

Péter Szeredi

There are several PROLOG applications where the speed of present implementations is insufficient. A solution for this problem is proposed in the framework of MPROLOG, the new Hungarian PROLOG implementation. Methods are presented for separating the critical parts of a program and introducing them as extra built-in procedures written in a lower level language. This way flexibility and very high level of PROLOG is still preserved for the application programmer, while program speed can be increased considerably. Experience gained in applying these methods to two programs in drug design and simulation is also presented.

A programming support environment for PROLOG program development

Péter Köves
Péter Szeredi

The Program Development Subsystem (PDSS) of the MPROLOG system is introduced. PDSS, which is itself written in PROLOG, supports the whole development process of a PROLOG module, featuring syntax driven input-output and editing functions and also sophisticated tools for running and testing the program including tracing, breakpoint and error handling. Plans for further development of PDSS are outlined.

abstracts

from Buenos Aires:

Towards Constructive Data Bases

Veronica Dahl

We discuss data bases that can be viewed as sets of specifications for combining different components into desired configurations or structures. These are constructed on demand, according to the combination rules stored and the particular requirements in a user's query. We emphasize intelligent synchronization of constructive processes, and relate our proposal to recent development on concurrent logic programming.

On Data Base Systems Development Through Logic

Veronica Dahl

We discuss the use of logic as a single tool for formalizing and implementing different aspects of data base systems in a uniform manner. The discussion focusses on relational data bases with deductive capabilities and very high level querying and defining features. The computational interpretation of logic is briefly reviewed, and then we examine several pros and cons concerning the description of data, programs, queries and language parser in terms of logic programs. While discussing the inadequacies, we show how to overcome them by introducing convenient extensions into logic programming. Finally, we present an experimental data base query system with a natural language front end, implemented in Prolog, as an illustration of these concepts. We include a description of the latter from the user's point of view and a sample consultation session in Spanish.

Translating Spanish Into Logic Through Logic

Veronica Dahl

We discuss the use of logic for natural language (NL) processing, both as an internal query language and as the programming tool. Some extensions to standard predicate calculus are motivated within the first of these roles. A logical system including them is informally described. It incorporates semantic as well as syntactic NL features, and its semantics in a given interpretation (or data base) determine the answer-extraction process. We also present a logic-programmed analyser that translates Spanish into this system. It equates semantic agreement with syntactic well-formedness, and can detect certain presuppositions, solve certain ambiguities and reflect relations among sets.

from Buffalo:

SNePSLOG A "Higher Order" Logic Programming Language

Stuart C. Shapiro, Donald P. McKay, João Martins, Ernesto Morgado

SNePSLOG is a logic programming interface to SNePS, the Semantic Network Processing System, and SNIP, the SNePS Inference Package. Assertions and rules written in SNePSLOG are stored as structures in a semantic network. SNePSLOG queries are translated into top-down deduction requests to SNIP. Assertions can also be stated in a way that triggers bottom-up SNIP deductions. Output from SNIP is translated into SNePSLOG formulas for printing to the user.

Since SNePSLOg predicates and formulas are represented as nodes in the semantic network, and since SNIP allows variables to range over any nodes, SNePSLOG expressions are not limited to first order predicate calculus. Examples in this paper will show SNePSLOG rules that quantify over predicates, and a SNePSLOG rule that has a rule in antecedent position which is treated like an atomic assertion during deduction.

Since SNIP supports several non-standard logical constants, the SNePSLOG syntax also allows them. Some of these are used below and will be explained where they first occur.

The examples in this paper refer to naval information.

from College Park, Maryland:

On Indefinite Databases and the Closed World Assumption

Jack Minker

A database is said to be indefinite if there is an answer to a query of the form Pa V Pb where neither Pa nor Pb can be derived from the database. Indefinite databases arise where, in general, the data consists of non-Horn clauses. A clause is non-Horn if it is a disjunction of literals in which more than one literal in the clause is positive.

Horn databases, which comprise most databases in existence, do not admit answers of the form Pa V Pb where neither Pa nor Pb are derivable from the database. It has been shown by Reiter that in such databases one can make an assumption, termed the Closed World Assumption (CWA), that to prove that Pa is true, one can try to prove Pa, and if the proof for Pa fails, one can assume Pa is true.

When a database consists of Horn and non-Horn clauses, Reiter has shown that it is not possible to make the CWA. In this paper we investigate databases that consist of Horn and non-Horn clauses. We extend the definition of CWA to apply to such databases. The assumption needed for such databases is termed the Generalized Closed World Assumption (GCWA). Syntactic and semantic definitions of generalized closed worlds are given. It is shown that the two definitions are equivalent. In addition, given a class of null values it is shown that the GCWA gives a correct interpretation for null values.

from Edinburgh:

Logic Programming: A Computing Tool for the Architect of the Future

Peter S. G. Swinson

(department of Architecture)

Computer Aided Architectural Design is reviewed with particular reference to new software techniques that are becoming available. The needs of the designer are examined leading to a specification for the computing tools that may serve architects in the future. The paper concludes by reporting on the results of early studies into one radically new tecnique — logic programming.

from London, England:

An Introduction to Logic Programming

K. L. Clark

In this paper we introduce the use of the Horn clause subset of predicate logic as a

programming language. We do this mainly through example programs. These highlight the novel aspects of logic programming, such as the ability to use the same program to compute a relation and its inverse, and the fact that it is suitable both for symbolic manipulation and deductive retrieval of information. The computation of a logic program is an inference. We briefly describe the inference procedure used by the PROLOG logic programming language. We also describe some of the extra logical features pf PROLOG and indicate how it might be used to implement expert systems.

Logic as a Database Language

Robert Kowalski

This paper investigates the application of logic to databases in the restricted sense which regards a database as a collection of assumptions expressed in symbolic logic. A database query is regarded as a theorem to be proved from the assumptions.

This approach contrasts with the interpretation of a database as a relational structure. Queries are expressed in logic but are answered by evaluating them in the structure. This is the approach which is employed in the relational database model.

The difference between the two approaches has been characterised by Nicolas and Gallaire as the difference between regarding a database as a *theory* and regarding it as an *interpretation*. When a database is regarded as a theory it is natural to describe data both by means of explicit assertions and by means of general rules. The inclusion of recursive definitions can be accomodated without leaving first-order logic. Such a database can have many models, i.e. interpretations which satisfy the theory. In this case the database can be regarded as describing incomplete information, since it does not distinguish a unique interpretation from among its many models. Although the logic itself is only two-valued, a yes-no query can be answered in one of three different ways: "Yes" if it can be proved, "No" if its negation can be proved, and "Don't know" if neither it nor its negation can be proved.

When a database is regarded as an interpretation it is natural to restrict database description to the explict enumeration of the tuples which belong to the relations in the database. Although general rules defining virtual relations can be incorporated into queries, recursive definitions cannot be included without leaving first-order logic. Since every yes-no query can only be answered only "yes" or "no", incomplete information cannot be dealt with without leaving two-valued logic.

Prolog as a Logic Programming Language

R. A. Kowalski

This paper is concerned with the relationship between logic programming and Prolog — and with the bearing this has on Prolog programming methodology. It is motivated in part by McDermott's advocacy of Prolog as a programming language for Artificial Intelligence and by his complaint that claims for its close relationship with logic are unjustified. Our position is in accordance with van Emden's reply to McDermott.

Although Prolog is best thought of as based on Horn clause logic, it incorporates only a simple backtracking proof procedure. Because of this limitation and, for other reasons, it provides various extralogic primitives to compensate. These can be used to simulate, rather awkwardly, programming styles more appropriate to conventional programming languages.

Our thesis is that the extralogical features of Prolog ought to be used in a disciplined fashion. Whenever possible, their use should be encapsulated in the definition of higher-level features which extend either the logical primitives of the language or its control facilities. Used in this way, the otherwise dangerous extralogic features of Prolog provide a useful safety valve which can be used to extend the power of the language while retaining its logical foundations.

---

from London, Ontario:

Logic and Programming Methodology

E. W. Elcock

(No abstract provided).

---

from Lexington, Kentucky:

Focalizers, the Scoping Problem, and Semantic Interpretation Rules in Logic Grammars

Michael C. McCord

This paper deals with a system of semantic interpretation for natural language within the framework of logic programming. Of special interest are a class of grammatical items called *focalizers,* and the problem of determining their scopes in logical form. Focalizers include quantificational determiners, certain adverbs, and abstract items relating to discourse structure.

Prolog as a Relationally Complete Database Query Language which Can Handle Least Fixed Point Operators

Derek J. Wright

The purpose of this paper is to bring together two fields: work done towards the design of "universal" relational database query languages and the, as yet, unrelated work done on the programming language PROLOG.

Classical, relationally complete query languages (the abstract relational algebra, relational calculus and the implemented "more than" relationally complete languages such as Sequel, Query by Example) are unable to handle the class of queries that involve least fixed points (LFP). Typical queries in this class are the "lowest common ancestor problem" and the "airline connecting flights problem".

PROLOG (PROgramming language based on LOGic), used mostly for AI applications, handles these queries in a natural, recursive manner where relations are manipulated as built-in datatypes.

PROLOG is shown to be a data independet, more than relationally complete query language that can handle LFP operations. The future of PROLOG as a possible "universal" query language is also discussed.

---

from Lisbon:

Selective Backtracking

Luís Moniz Pereira
António Porto

In this paper we review selective backtracking and address general implementation issues.

---

from Lisbon and Leuven:

Revision of Top-Down Logical Reasoning Through Intelligent Backtracking

Maurice Bruynooghe
Luís Moniz Pereira

First, we develop a theory for a more intel-

ligent form of backtracking, in sequential or parallel to-down executions of Horn clause logic programs, which exploits the relationships among states of the search. The theory allows more flexible and efficient backtracking strategies, showing how to avoid backtracking to alternatives which, a priori, cannot possibly prevent subsequent repetition of the same failures. The theory also shows which parts of the derivation state may be kept on backtracking. It thus provides a powerful method for revision of top-down logical reasoning.

Second, we introduce simplifying assumptions which have lead to a practical application of the theory, in the form of an intelligently backtracking interpreter of Prolog programs.

from Lyngby, Denmark:

A Notion of Grammatical Unification Applicable to Logic Programming Languages

Jan Maluszynski
&
Jørgen Fischer Nilsson

This paper presents an extension of the notion of unification applied in theorem proving and logic programming. Potential use of the extended unification in logic programming is surveyed. A theorem relating to the construction of a most general extended unifier is formulated and proved. Finally, a unification algorithm is suggested.

from Marcoussis, France:

Impacts of Logic on Data Bases

H. Gallaire

This paper deals with the relationships between logic and relational data bases. A goal of the paper is to show, through many results published in the literature, how logic can provide a formal support to study classical database problems, and in some cases, how logic can go further, helping first in their comprehension, and then in their solution. The main points discussed are: query evaluation and data base schema analysis.

Key words: relational data bases, logic, query languages, integrity constraints, deductive data bases, query evaluation, dependencies, data base schema analysis.

Metalevel Control for Logic Programs

Hervé Gallaire
Claudine Lasserre

This paper deals with means of incorporating metaknowledge capabilities to a PROLOG-like Horn clauses interpreter, be they general control strategies or specific ones. The paper gives an overview of current approaches and introduces new means of control. A solution is investigated, consisting of a metalevel expression of control of a standard interpreter. Actions allowed by the metarules are described and an interpreter for the extended language is sketched.

from Marseille:

Prolog and Infinite Trees

A. Colmerauer

The paper deals with the manipulation of infinite trees in the context of the programming language Prolog. With this purpose a novel and concise model of Prolog is presented. The model does not explicitly involve the first order logic. The problem of unifying two terms is replaced by that of determining whether or not a system of equations has at least one solution. Several examples of the use of infinite trees are also given.

from Prague:

Horn Clause Programs and Recursive Functions Defined by Systems of Equations

Jan Sebelik

Every recursive function can be defined by a system of equations. At the first Logic Programming Workshop, Debrecen, Hungary, July 1980, R. Kowalski raised the question of interpretting these systems by means of Horn Logic.

In the paper, there is described an algorithm which transforms every such a system of equations into a Horn clause program computing the same function. On the other hand, every Horn clause program the predicate symbols of which are all at least unary can be transformed into a system of equations defining the same function.

In the end of the paper, there are discussed certain syntactic properties of Horn clause programs obtained from systems of equations. Finally, a syntactic condition for Horn clause programs computing primitive recursive functions is formulated.

from Santa Monica:

Logic Programming in DADM

Charles Kellog

The DADM (Deductively Augmented Data Management) system (see Kellog and Travis [1981]) has been developed over the past few years as a system for providing deductive access to large external DMS's and their stores of extensional data. The DADM deductive processor, therefore, has beeen specifically designed to support deductive question answering rather than logic programming.

However, DADM is a complete first order logic system and it has certain features in its Logic and Control components that can be used to advantage for logic programming purposes. We discuss and illustrate several of these features after briefly reviewing some of the salient general characteristics of the system.

from SRI:

Problems in Logical Form

Robert C. Moore

Most current theories of natural-language processing propose that the assimilation of an utterance involves producing an expression or structure that in some sense represents the literal meaning of the utterance. It is often maintained that understanding what an utterance literaly means consists in being able to recover such a representation. In philosophy and linguistics this sort of representation is usually said to display the *logical form* of an utterance.

This paper surveys some of the key problems that arise in defining a system of representation for the logical forms of English sentences and suggests possible approaches to their solution. We first look at some general issues relating to the notion of logical form, explaining why it makes sense to definbe such a notion only for sentences in context, not in isolation, and we discuss the relationship between research on logical form and work on knowledge representation in artificial inte-

abstracts

lligence. The rest of the paper is devoted to examining specific problems in logical form. These include the following: quantifiers; events, actions and processes; time and space; collective entities and substances; propositional attitudes and modalities; questions and imperatives.

Automatic Reduction for Commonsense Reasoning: An Overview

Robert C. Moore

How to enable computers to draw conclusions automatically from bodies of facts has long been recognized as a central problem in artificial-intelligence (AI) research. Any attempt to address this problem requires choosing an application (or type of application), a representation for bodies of facts, and methods for deriving conclusions. This article provides an overview of the issues involved in drawing conclusions by means of deductive inference from bodies of commonsense knowledge represented by logical formulas. We first briefly review the history of this enterprise: its origins, its fall into disfavor, and its recent revival. We show why applications involving certain types of incomplete information resist solution by other techniques, and how supplying domain-specific control information seems to offer a solution to the difficulties that previously led to disillusionment with automatic deduction. Finally, we discuss the relationship of automatic deduction to the new field of "logic programming", and we survey some of the issues that arise in extending automatic-deduction techniques to nonstandard logics.

---

from Syracuse:

Amalgamating Language and Metalanguage in Logic Programming

Kenneth A. Bowen
Robert A. Kowalski

It is argued that present-day logic programming systems exhibit shortcomings which can be overcome by extending the original object language to include that portion of the metalanguage which deals with the object language provability relation. Such a system is sketched, and some of its applications and properties are presented.

≠ ≠ ≠

Logic programming systems have proven to be a powerful tool for computer science. These systems have been especially congenial for work in artificial intelligence and database management (cf. Gallaire and Minker [1978], and Kowalski [1979]). Inevitably, as with almost any system, shortcomings have been discovered. Central among these is the problem of managing the system's database of clauses. Simply put, it is this: The conceptual basis of logic programming is deduction from a *single fixed* theory, while many applications must deal with deduction from *varying* or *alternative* theories. Thus in maintenance over time of a simple relational database, tuples are added, deleted, or modified. Addition of a tuple to a relation corresponds to assertion of a simple unit clause. Here only one theory at a time is involved, but it changes over time. Moreover, the problem of maintaining integrity constraints amounts to testing the consistency of the proposed addition with the theory constituting the database.

The need in this example is for an ability to explicitly refer to theories (i.e., collections of clauses) and to discuss derivability from these theories. Our approach is to construct a system which amalgamates an object-level logic system with a portion of a metalanguage suitable for formalizing the derivability relation of the original object language system. We shall argue that the resulting system will have greater expressive and problem-solving power than the original object language system alone. However, we will carry out this amalgamation in such a way as to preserve the standard semantics of logic. Our purpose in this enterprise is primarily practical. However, in the resulting system it is possible to carry through rather direct proofs of incompleteness phenomena of the sort first discovered by Godel [1931]. Our use of metalanguage is similar to that of Weyhrauch [1980], the main difference being that he does not consider systems, such as ours, which completely amalgamate object and metalanguage.

---

from Uppsala:

Uniform — A Language based upon Unification which unifies (much of) Lisp, Prolog, and Act 1

Kenneth M. Kahn

Uniform is an AI programming language under development based upon augmented unification. It is an attempt to combine, in a simple coherent framework, the most important features of Lisp, actor languages such as Act I and SmallTalk, and logic programming languages such as Prolog. Among the unusual abilities of the language is its ability to use the same program as a function, an inverse function, a predicate, a pattern, or a generator. All of these uses can be performed upon concrete, symbolic, and partially instantiated data. Uniform features automatic inheritance from multiple super classes, facilities for manipulation of programs, a limited ability to determine program equivalence, and a unification-oriented database.

A Programming Language Based on a Natural Deduction System

Sten-Åke Tärnlund

We shall take up a programming language based on natural deduction. Our presentation starts with the subset of Horn clause logic on which Prolog is based but continues with inference rules that provides a programming language on full predicate logic. We can treat negation at the object level, virtual classes, identity that gives the notion of functions which helps us to prove termination on infinite data structures (streams). Moreover, we have several computation rules, e.g., a demand driven rule, an instantiation rule.

---

from Yale:

Inductive Inference of Theories From Facts

Ehud Y. Shapiro

This paper is concerned with model inference problems and algorithms. A *model inference problem* is an abstraction of the problem faced by a scientist, working in some domain under some fixed conceptual framework, performing experiments and trying to find a theory capable of explaining their results. In this abstraction the domain of inquiry is the domain of some unknown model *M* for a given first order language *L*, experiments are tests of the truth of sentences of *L* in *M*, and the goal is to find a set of true hypotheses that imply all true testable sentences.

The main result of this paper is a general, incremental algorithm for solving model inference problems, which is based on the Popperian methodology of conjectures and

refutations [Popper 59, Popper 68]. The algorithm can be shown to identify in the limit [Gold 67] any model in a family of complexity classes of models, it is the most powerful of its kind, and is flexible enough to have been successfully implemented for several concrete domains.

This model inference algorithm has two tunable parameters: one determines how complicated the structure of hypotheses is; the other, how complex derivations from the hypotheses can be. Together they determine the class of models that can be inductively inferred in the limit by the algorithm. On the one hand they can be set so that the model inference algorithm can identify in the limit any model with complexity bounded by any fixed recursive function. On the other hand they can be set so that the algorithm, appropriately implemented, can infer axiomatizations of concrete models from a small number of facts in a practical amount of time. The performance of the *Model Inference System* demonstrates this.

The Model Inference System is based on this model inference algorithm, specialized to infer theories in Horn form. It has been implemented in the programming language Prolog [Pereira et al. 78]. As an example, in the domain of arithmetic, the system inferred the set of axioms described in Figure 1-1 below from 36 facts in 27 seconds CPU time. The system has discovered an axiomatization for dense partial order with end points. It has successfully synthesized logic programs [Kowalski 79a] for simple list-processing tasks such as append, reverse and most of the examples described in Summers' thesis [Summers 76]. It has also synthesized logic programs for satisfiability of boolean formulas, binary tree inclusion, binary tree isomorphism and others.

As part of the general algorithm, an algorithm for backtracing contradictions was discovered. This algorithm is applicable whenever a contradiction occurs between some conjectured theory and the facts. By testing a finite number of ground atoms for their truth in the model the algorithm can trace back a source for this contradiction, namely a false hypothesis, and can demonstrate its falsity by constructing a counterexample to it. The existence of such an algorithm seems to be relevant to the philosophical discussion on the refutability of scientific theories [Harding 76], and specialized to Horn theories may be a practical aid for the debugging of logic programs.

An Algorithm that Infers Theories from Facts

Ehud Y. Shapiro

This paper is an informal summary of the results described in the report mentioned above.

Algorithmic Program Debugging

Ehud Y. Shapiro

The notion of program correctness with respect to an interpretation is defined for a class of programming languages. Under this definition, if a program terminates with an incorrect output then it contains an incorrect procedure. Algorithms for detecting incorrect procedures are developed.

A logic program implementation of these algorithms is described. Its performance suggests that the algorithms can be the backbone of debugging aids that go far beyond what is offered by current programming environments.

Applications of algorithmic debugging to automatic program construction are explored.

research centres addresses

Here are some more:

Department of Computer Science
Building 344 and 343
Technical University of Denmark
DK-2800 Lyngby Denmark

UPMAIL
Compter Science dept.
Uppsala University
Sturegaten 4A
S-75223 Uppsala Sweden

Department of Computer Science
Yale University
New Haven, CT 06520 USA

EdCAAD Studies
University of Edinburgh
20 Chambers St.
Edinburgh EH1 16Z UK

Centro de Informática
Laboratório Nacional de Engenharia Civil
1799 Lisboa Portugal

Dept. of Computer Science
University of Western Ontario
London, Ontario N6A 5B9 Canada

Dept. of Computer Science
University of Maryland
College Park, MD 20742 USA

Dept. of Cybernetics and O.R.
Charles University
Malostransk Namesti 25
Praha 1 — 11800 Czechoslovakia

Dept. of Computer Science
State University of New York at Buffalo
Amherst, NY 14226 USA

Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025 USA

System Development Corporation
2500 Colorado Ave.
Santa Monica, CA 90406 USA

Depto. de Matematica
Faculdad de Ciencias Exactas
Ciudad Univ. Nunez, Pab. I
1428 Buenos Aires Argentina

Dept. of Computer Science
Yale University
P.O.Box 2158
10 Milhouse Ave.
New Haven, CT 6520 USA

Laboratoires de Marcoussis — C. G. E.
Route de Nozay
91460 Marcoussis France

Iowa State University
Dept. of Computer Science
Ames, IA 50011 USA

Dept. of Computer Science
University of Kentucky
Lexington, KY 40506 USA

PERSONAL CHANGES OF ADDRESS

1. *David Warren is now with SRI International, 333 Ravenswood Ave., Menlo Park CA 94025, USA.*
2. *Chris Mellish is now with Cognitive Studies Programme, Arts Building, University of Sussex, Brighton BN1 9QN, UK.*
3. *William Clocksin is now with the Robot Welding Project, Dept. of Engineering Science, University of Oxford, Parks Rd, Oxford OX1 3PJ, UK.*
4. *Bob Welham is now with Dept. of Computer Science, University of Strathclyde, Livingstone Tower, 26 Richmond St., Glasgow G1 1XH, UK.*
5. *Derek Wright is now with Dept. of Computer Science, University of San Francisco, San Francisco, CA 94117, USA.*

CORRECTIONS TO No. 1

1. *We apologize to John S. Conery for having his name mispelled as John S. Coery.*
2. *"Programming in Prolog", the primer by W. F. Clocksin and C. Mellish, has been published in September by Springer Verlag, and not by Edinburgh University Press as advertised. A paperback, it has 296 pages and costs approximately US $16.00 or D.M. 35,00.*