PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

Directed Relations and Inversion of Prolog Programs

Yoav Shoham and Drew V. McDermott

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520, USA

Abstract

We suggest that Prolog predicates should be viewed as denoting *directed relations*, the direction being a set of partitions of the variables occurring in it. *Functions* are a special case of directed relations with a direction that contains a single partition. *Complete* relations are those whose direction includes all partitions of the variables.

The paper explores some consequences of such a realistic view of Prolog. We discuss the issue of extending the directionality of a relation and in particular investigate ways of inverting functions mechanically. Three algorithms for function inversion are given and their performance on nontrivial problems, as well as their shortcomings, are demonstrated.

Finally we present an interactive system that traverses a directed compuatation tree, which is a computation tree in which with each node we associate information about the variables appearing in it, and demonstrate its performance.

1 Introduction

The paradigm of logic programming requires the view of procedure declarations as logical formulae, and in the case of Prolog ([Kowalski 74], [Clocksin & Mellish 81]) these formulae are restricted to Horn clauses. The various formal semantics of logic programming provided in [Van Emden & Kowalski 76] define the (never implemented) "pure" Prolog in the spirit of [Robinson 65]. The inapplicability of these semantics is made painfully clear to the novice user of any existing Prolog implementation. A recent attempt has been made to define realistic formal semantics for Prolog [Jones & Mycroft 83], and we expect to see further formal discussion of the issue in the future.

We too are advocating what seems to us as a realistic view of Prolog programs. The formal part of the presentation is short and is intended mainly as a motivation for the rest of the discussion which describes experimental techniques and initial practical results. Our position is that Prolog predicates do not denote *relations* but rather what we term as *directed relations*, which we define in section 2 as an obvious generalization of *functions*. Grabbing the bull by the horns in this way, we explore ways of extending the directionality of the predicates. The bulk of the paper is section 3 which deals the with the special case of function inversion. Section 4 addresses the general problem of exploring a *directed computation tree*, which is a computation tree in which with each node we associate two lists - the variables which are bound when we enter the node and the variables that are bound when we exit the node. In the last two sections we survey some of the related literature and summarize the main points made in the paper.

2 Directed Relations

Consider the familiar Quicksort, defined by, say:

```
qsort([H|T],S) :-
 split(H,T,A,B),!,
 qsort(A,A1),
 qsort(B,B1),
 append(A1,[H|B1],S).
qsort([],[]).

split(H,[A|X],[A|Y],Z) :-
 order(A,H), split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :-
 not(order(A,H)), split(H,X,Y,Z).
split(_,[],[],[]).

order(A,B) :- A<B.
```

One would expect invocation of the goal `qsort(X,[1,2,3])` to bind X successively to all six permutations of [1,2,3]. What in fact will happen is that the interpreter will return two error messages and fail. Worse still, consider the following definition of Insertionsort:

```
insort([],[]).
insort([X|L],M) :-
 insort(L,N),insert(X,N,M).
```

```
insert(X,[],[X]).
insert(X,[A|L],[X,A|L]) :-
 order(X,A),!.
insert(X,[A|L],[A|M]) :-
 insert(X,L,M).
```

where **order** is defined as above. When the goal **insort(X,[1,2,3])** is invoked the interpreter diverges after yielding one permutation, displaying an infinite number of error messages. A similar call of an appropriately defined Bubblesort diverges immediately and another sort we defined resulted in a circular list.

The problem is obviously that goals are invoked with the "wrong" arguments instantiated, or in the wrong "mode" to use Prolog-10 terminology. In this case we might say that **sortname(X,Y)** is a function[2] from X to Y rather than a relation on X and Y. More generally one can make the following definitions:

> **Definition:** Let $R(X_1,...,X_n)$ be an n-ary Prolog predicate with an intended interpretation $I=\{<I_1^i,...,I_n^i>\}$, and let $V=\{X_1,...,X_n\}$. R is said to be a function from from V1 to V2 if $<V1,V2>$ is a partition of V, and for all instantiations of V1 (and thus partial instantiations of V) invoking R will fairly generate all the tuples in I unifiable with V.
> For our purposes a partition of a set S is a tuple $<S1,S2>$ of disjoint sets whose union is S. A fair generation of a sequence is one in which any given element is generated after a finite amount of time.
>
> Example: **qsort(X,Y)** is a function from [X] to [Y] and from [X,Y] to [], but not from [Y] to [X].
>
> **Definition:** A Prolog predicate R with a given intended interpretation is said to be a D-directed relation if D is a set of tuples $\{<V1_i,V2_i>\}$ such that R is a function from $V1_i$ to $V2_i$ for all i. Note that a function from V1 to V2 is a special case of a directed relation, one that is $\{<V1,V2>\}$-directed.
>
> Example: **qsort(X,Y)** is $\{<[X,Y],[]>,<[X],[Y]>\}$-directed.
>
> **Definition:** A Prolog predicate R is called complete if it is D-directed for D the set of all partitions of the set of variables in R.

[2]Since our formalization serves mainly to provide intuition for the remainder of the paper, we allow ourselves some freedom in using the terminology. As we will define the term *function* it will always denote a nondeterminstic function.

It is not immediately clear what the direction of a particular predicate in a program is - the traditional view encourages regarding it as complete, while typically it is written as a function. However once a predicate is identified as a function a question that arises naturally is whether its directionality can be extended, perhaps even so as to make it complete (in the latter case we will say that the predicate had been completed). A special case is where the directed relation is a function from V1 to V2, and we want to extend it to be $\{<V1,V2>,<V2,V1>\}$-directed, that is we want to invert the function. Section 3 deals with function inversion, and section 4 deals with the more general question of determining the directionality of a predicate.

3 Function inversion

The general problem of function inversion is hard and suggests some immediate caveats. For example a solution to the general problem would yield a factoring algorithm and a statement on Fermat's last theorem. In general automating the inversion of number theoretic functions is problematic - such a process would have to rely on a detailed representation of mathematical objects, which Prolog (like any other programming language) lacks. That is not to say that engaging in such a task is a wasted effort, only that such an effort will center around representation issues (cf. [McAllester 83], [Lenat 82]). In fact our original motivation was to invert the knowledge of solving counting problems in combinatorics into knowledge of proving combinatorial equalities (see below). What we do in this section is provide some simple inversion procedures and begin to explore their properties. The flavor of the presentation is empirical - the reader should expect demonstration of the procedures' power rather than a thorough theoretical analysis.

We first present a simple inversion algorithm which stated roughly says "Given a conjunctive goal solve the conjuncts in reverse order. Given a single goal reduce it if possible, otherwise execute it".

Algorithm 1: A simple inversion

```
invgoal((A,B)) :-
  !,invgoal(B),invgoal(A).
invgoal(A) :-
  clause(A,B),invgoal(B).
invgoal(A) :-
  not(clause(A,_)),call(A).
```

When we apply the above algorithm to the sorting programs from section 2 we observe the following behavior:[3]

[3]All the examples in this paper were done on a DEC20 running Prolog-10 version 3.47.

Example 1: inverting Quicksort and Insort

```
| ?- invgoal(qsort(X,[1,2,3])).
X = [1,2,3] ;
X = [1,3,2] ;
X = [2,1,3] ;
X = [2,3,1] ;
X = [3,1,2] ;
X = [3,2,1] ;
no
| ?- invgoal(insort(X,[1,2,3])).
X = [1,2,3] ;
X = [1,3,2] ;
X = [2,1,3] ;
X = [2,3,1] ;
X = [3,1,2] ;
X = [3,2,1] ;
no
| ?-
```

which is indeed what is required.

It is surprising (at least to us) that a simple procedure such as Algorithm 1 proves effective in these nontrivial cases. Why does it work?

A Prolog program (by which we mean a list of definite clauses, see [Apt & Van Emden 82]) and a goal define an AND-OR computation tree. The Prolog interpreter traverses this tree depth-first from left to right.

Fact: Algorithm 1 simulates the Prolog interpreter, preserving the depth-first strategy and the left-to-right traversal at the OR nodes, but traversing the AND nodes right-to-left.

Since any traversal of the computation tree represents a sequence of steps in the resolution process ([Robinson 65]), any traversal of the tree constitutes a sound computation. This together with the previous fact establish the soundness of Algorithm 1. Of course the more interesting question revolves around its completeness - is it guaranteed to invert any function? The answer is no, and we demonstrate it shortly. First however we consider the cases where it does work. Like we said earlier we will not present a rigorous analysis, but will briefly give some intuition.

Consider a program P and a predicate R. All (partial) instantiations of variables ocurring in R define an AND-OR computation tree which is in general infinite. There are two reasons why a successful goal may fail or diverge when the input-output status of its variables is changed. One is that the structure of the induced computation tree is changed, that is some OR-nodes have a different number of sons than previously (instantiating a previously unbound variable may reduce the number of sons and vice versa). In particular the new tree may have a new infinite path; we will see such an example shortly. The other reason has to do with Prolog's special features. In the Quicksort example the feature was numerical comparison ("<"), which requires its two arguments to be instantiated to integers. Algorithm 1 Only deals with the second kind of problem, and is heuristic in nature. It assumes that the procedure being inverted is "backwards deterministic", to use Dijkstra's terminology [Dijkstra 83].

Algorithm 1 is a bit simplistic in that it only reverses the order of computation. The following procedure adopts the same basic algorithm, but pays more respect to special Prolog features.

Algorithm 2: A less simple inversion

```
invgoal(invgoal(X)) :- call(X).

invgoal(assert(X)) :- retract(X).
invgoal(retract(X)) :- assert(X).

invgoal(A is B+C) :- var(B),B is A-C.
invgoal(A is B+C) :- var(C),C is A-B.
invgoal(A is B-C) :- var(B),B is A+C.
invgoal(A is B-C) :- var(C),C is B-A.
invgoal(A is -B) :- B is -A.
% and any other mathematical
% inversions which are needed

invgoal((A,B)) :-
  !,invgoal(B),invgoal(A).
invgoal(A) :-
  clause(A,B),invgoal(B).
invgoal(A) :-
  not(clause(A,_)),call(A).
```

Armed with this slightly more meaty algorithm we can do some more inversions. The next example brings us back to our original motivation, that of inverting the solution of counting problems in combinatorics. Since the example is not trivial, and because we think automating the solution of problems in combinatorics is of interest in itself, this example will be a bit long and the reader's indulgence is requested. In [Shoham 84] we describe a program (FAME I) for proving combinatorial equalities by combinatorial arguments. The general structure of proving an two expressions equal by a combinatorial argument is showing that both are a correct solution to the same counting problem. An example of an equality is N*c(N-1,R-1)=R*c(N,R), where c(X,Y) stands for "X choose Y". An example of a combinatorial proof of this equality is that both describe the number of ways to choose a team of R players from N candidates and appoint a captain from among them. The first expression describes the process of first choosing the captain and then the rest of the team, and the second expression describes the process of first choosing the whole team and then the captain. In that paper we pointed out the shortcomings of our program, namely that the knowledge of counting was only implicit in it and there was no obvious way to gracefully extend the program to handle other problems in combinatorics. The "correct" way to go about it, we said, was to write a program (FAME II) that solved counting problems. Then another program could be written that used the knowledge of FAME II to synthesize a program similar to FAME I, by inverting the knowledge of counting.

The following is an example of a solving a problem by FAME II (translated into English it reads "In how many ways can you choose a set **set2** of size r from a set **set1** of size n, and choose a set **set3** of size 1 from **set2**?").

```
| ?- count(
|      [(set1,n),(set2,r),(set3,1)],
|      [subset(set3,set2),
|       subset(set2,set1)],
|      Solution).
Solution = c(r,1)*c(n,r)
yes
| ?-
```

We now ask the converse question - "What counting problem is the expression **c(n,r)*c(r,1)** a solution to" by inverting **count**:

Example 2: inverting Count

```
| ?- invgoal(
|      count(X,Y,c(n,r)*c(r,1))).
** Error: evaluate(_246)

X = [(_241,r),(_368,1),
     (_242,n)|_832],
Y = [subset(_241,_242),
     subset(_368,_241)] ;

X = [(_369,r),(_368,1),
     (_242,n),(_241,r)|_948],
Y = [subset(_241,_242),
     subset(_368,_369)]

yes
| ?-
```

Notice that these two solutions are correct, and the most general - X may contain an arbitrary number of (set,cardinality) tuples, but Y is restricted to exactly the two above subset relations.

The code for **count** is too long to include here. To give the reader a better feel for the two algorithms consider the following definition of **abs**:

```
abs(N,M) :- N<0,!,M is -N.
abs(N,N).
```

Given the goal **abs(X,2)** Algorithm 2 will execute as follows:

Example 3: Inverting abs

```
| ?- invgoal(abs(X,2)).
X = -2 ;
** Error: evaluate(_31)
X = 2 ;
no
| ?-
```

while Algorithm 1 will only return the second (positive) answer.

The last example can also serve to demonstrate the effect of the cut sign on invertibility. On the one hand notice that although one of the clauses of **abs** contains a cut Algorithm 2 returned both answers. The less happy news is the following:

Example 4: The perils of !.

```
| ?- invgoal(abs(X,-2)).
** Error: evaluate(_31)
X = -2 ;
no
| ?-
```

The reason for this "error" is the use of the cut symbol to improve efficiency. The way to eliminate this bug is to change the definition of **abs** to:

```
abs(N,M) :- N<0,M is -N.
abs(N,N) :- not(N<0).
```

and so the immediate lesson is that discipline is required in defining a function that is to be invertible.

The next algorithm, Algorithm 3, may seem at first sight like an elaborate version of Algorithm 2. It has two phases - in the first interactive phase the system inverts functions, asserts their inverse to the database and writes them to a file - all according to the user's specification. In the second independent phase the inverted code is simply run.

As it is presented here, the inverse of a function **F** is called **inv(F)**. The algorithm traverses the AND-OR like the two previous algorithms tree and whenever a goal A' is unifiable with a head of a clause **A :- B**, the user is given the choice of continuing along that branch of the tree or quitting it. Continuing means asserting the clause **inv(A) :- inv(B)**, and recursing on **B**. This is in contrast to the previous algorithm where if a goal is unifiable with a head of a clause the algorithm will definitely recurse on the body of that clause. The advantage of Algorithm 3 is that the user can detect infinite recursion during the inversion phase, and prevent it from occurring during runtime. In this way the user can cope with the first problem mentioned earlier - the change in the AND-OR tree structure and in particular the introduction of new infinite paths. The disadvatage is that when the user decides to quit pursuing a branch of the tree he may lose information. The example we give is the inversion of a function with side effects. The predicate **gensym** is defined in [Clocksin & Mellish 81] (p. 150), and since our definition is very similar we will not repeat it here. The reader is reminded that **gensym('string',S)** binds S to **'string'** concatenated with the ASCII representation of the (global) number associated with 'string', and that number is incremented.

Algorithm 3: interactive inversion

Phase I: findinv

```
findinv(X) :-
 nl,write('Do you want the resulting code
      asserted in the database? (y/n) '),
 nl,read(A),nl,
 write('(Where) do you want to save the
       resulting code? (filename/none)'),
 nl,read(F),
 findinv(X,A,F).

findinv(X,A,none) :- !,findinv1(X,A,no).
findinv(X,A,F) :-
 tell(F),findinv1(X,A,yes),told.

findinv1([A|B],X,Y) :-
 !,findinv1(A,X,Y),findinv1(B,X,Y).
findinv1([],_,_) :- !.

findinv1((A,B),X,Y) :-
 !,findinv1(A,X,Y),findinv1(B,X,Y).

findinv1(A,X,Y) :-
 telling(F),tell(user),nl,
 writel(['Do you want to invert the
          goal ',A,'? (y/n)']),nl,
 told,tell(F),
 read(n),!.

findinv1(A,X,Y) :-
 clause(A,B),invclause(A,B,X,Y),fail.
findinv1(_,_,_).

invclause(A,B,X,Y) :-
 invbody(B,C),invassert(A,C,X),
 invwrite(A,C,Y),findinv1(C,X,Y).

invbody((X,Y),Z) :-
 !,invbody(X,X1),invbody(Y,Y1),
   andappend(Y1,X1,Z).
invbody(X,X).

andappend((A,B),C,D) :-
 !,andappend(A,(B,C),D).
andappend(A,B,(A,B)).

invassert(_,_,n) :- !.
invassert(A,C,_) :-
 asserta((inv(A):- inv(C))).

invwrite(A,C,no) :- !.
invwrite(A,C,yes) :- nl,
 writel(['inv(',A,') :- inv(',C,').']).
```

Phase II: inv

```
inv(inv(X)) :- call(X).
inv((A is B+C)) :- B is A-C.
inv((A is B+C)) :- C is A-B.
% and other math inversions

inv(assert(X)) :- retract(X).
inv(retract(X)) :- assert(X).

inv((A,B)) :- !,inv(A),inv(B).
inv(A) :- not(clause(A,_)),A.
```

Example 5: inverting gensym

```
| ?- findinv(gensym(X,Y)).

Do you want the resulting code asserted
   in the database? (y/n) |: y.
(Where) do you want to save the
   resulting code? (filename/no) |: no.
Do you want to invert the goal
   gensym(_31,_52)? (y/n) |: y.
Do you want to invert the goal
   name(_52,_219)? (y/n)  |: y.
Do you want to invert the goal
   append(_217,_218,_219)? (y/n) |: y.
Do you want to invert the goal
   true? (y/n)  |: n.
Do you want to invert the goal
   append(_513,_218,_515)? (y/n)  |: n.
Do you want to invert the goal
   integername(_216,_218)? (y/n)  |: y.
Do you want to invert the goal
   integername(_216,[],_218)? (y/n)|: y.
Do you want to invert the goal
   _626 is _216+48? (y/n)  |: y.
Do you want to invert the goal
   !? (y/n)  |: n.
Do you want to invert the goal
   _216<10? (y/n)  |: n.
Do you want to invert the goal
   integername(_627,[_629],_218)?
   (y/n)  |: n.
Do you want to invert the goal
   _628 is _216 mod 10? (y/n)  |: n.
Do you want to invert the goal
   _627 is _216/10? (y/n)  |: n.
Do you want to invert the goal
   name(_31,_217)? (y/n)  |: n.
Do you want to invert the goal
   getnumgensym(_31,_216)? (y/n)  |: y.
Do you want to invert the goal
   asserta(gensymnum(_31,_216))?
   (y/n)|: n.
Do you want to invert the goal
   _216 is _619+1? (y/n)  |: n.
Do you want to invert the goal !? (y/n)
|: n.
Do you want to invert the goal
   retract(gensymnum(_31,_619))?
   (y/n)|: n.
Do you want to invert the goal
   asserta(gensymnum(_31,1))?
```

```
   (y/n)|: n.

X = _31,
Y = _52

yes
| ?- inv(gensym(X,input7)).

X = input

yes
| ?-
```

Algorithm 2 will fail to invert **gensym**:

```
| ?- invgoal(gensym(X,input7)).
** Error: evaluate(_562)
! more core needed
[ Execution aborted ]

| ?-
```

Finally, we demonstrate that even when taken together the above algorithms will not suffice to invert all functions. Consider the following program:

```
f([a|X]) :- g(X).
f([b|X]) :- g(X).
g([c,_]).
g(X) :- f(X).
```

Considered as a function from [X] to [], f(X) acts as recognizer for the regular language $(a+b)*.c.\Sigma*$. Inverting f would cause it to act as a "fair" generator of the same language (in the sense defined in section 2). The reader should convince himself that none of the above algorithms will invert f. The reason for this is that with its argument instantiated, f defines a finite tree. With its argument uninstantiated it defines an infinite tree - there are an infinite number of OR-nodes that are roots of two infinite subtrees each (corresponding to the first two clauses). Neither the regular interpreter nor the algorithms presented will traverse both infinite subtrees of any OR-node.

At this point we should mention an obvious non-solution to all inversion problems (and predicate redirection in general) - conduct a breadth-first search of the computation tree. Both aspects of its "non-solutioness" (namely, its theoretical completeness and impracticality) can be demonstrated on the above program. We have implemented a breadth-first theorem-prover in Prolog; Invoking the goal bf(G) will initiate such a proof.[4]

Example 6: Generating $(a+b)^*.c.\Sigma^*$

```
| ?- bf(f(X)).

X = [a,c,_312] ;
X = [b,c,_516] ;
X = [a,a,c,_1342] ;
X = [a,b,c,_1818] ;
        .
        .
        .
X = [b,b,a,a,c,_14865] ;
X = [b,b,a,b,c,_15398] ;
X = [b,b,b,a,c,_15941] ;
! more core needed
[ Execution aborted ]

| ?-
```

We actually have another implementation of **bf** that, using side effects, uses space more economically. The pricei s in time - it took ten minutes clock time to generate 90 strings.

4 Exploring directions of relations

In the previous section we dealt with the problem of automatically or semi-automatically augmenting the directions of directed relations of a particular kind in a particular way, namely inverting functions. However, while we may not know how to invert a {<[X,Y,Z],[A,B,C]>}-directed relation (i.e., a function) we may be able to augment its direction to {<[X,Y,Z],[A,B,C]>,<[X,Y,C],[A,B,Z]>}. Furthermore, while we may not know how to invert either the <S1,S2>-directed relation R or the <S3,S4>-directed R, we may be able to actually complete the {<S1,S2>,<S3,S4>}-directed R (for example, if S1=S4=[X] and S2=S3=[Y] ...). Since the simpler function inversion problem is already complicated enough, we have not attempted to automate the general problem of predicate redirection. What we have opted for is an interactive program that, directed by the user, explores a *directed computation tree*. By that we mean an AND-OR computation tree, where with each node we associate two lists - the variables that are instantiated when the node is reached from its parent and the variables that are instantiated by the time the algorithm returns from the node to its parent (obviously the latter will be a superset of the former). Exact definition of the latter is a bit problematic, because different refutations may instantiate different variables. In this program we adopt an optimistic view and take the union of all such variables. It is trivial to take their intersection (see the predicate collect0 below), and possible to implement a more sophisticated procedure; our experience suggests that this is not a crucial issue.

Before we give the algorithm, we demonstrate its behaviour on the Quicksort example from the previous sections. **iowalk(I,G)**[5] will initiate a preorder depth first search of the directed computation tree whose root is <I,O,G>.

[4] The exact definition of bf is omitted for lack of space

[5] Here we borrow from the terminology of data dependency [McDermott 83]

Example 7: Walking the directed tree for Quicksort

```
| ?- iowalk([Y],qsort(X,Y)).
Do you want to pursue the goal:
   qsort(_52,_29)
   with the instantiated variables [_29] ?
|: y.

The IO relations in its subgoals are
The goal(s): true with [] as input
The goal(s):
  split(_480,_481,_482,_483),!,
  qsort(_482,_484),qsort(_483,_485),
  append(_484,[_480|_485],_479)
                with [_479] as input

Do you want to pursue the goal: true
   with the instantiated variables [] ?
|: n.
Do you want to pursue the goal:
   split(_480,_481,_482,_483)
   with the instantiated variables [] ?
|: y.

The IO relations in its subgoals are
The goal(s): true with [] as input
The goal(s): order(_1114,_1115),
  split(_1114,_1116,_1117,_1118)
                    with [] as input
The goal(s): order(_1123,_1124),
  split(_1124,_1125,_1126,_1127)
                    with [] as input

Do you want to pursue the goal: true
   with the instantiated variables [] ?
|: n.
Do you want to pursue the goal:
   order(_1114,_1115)
   with the instantiated variables [] ?
|: y.

The IO relations in its subgoals are
The goal(s): _1674<_1675 with [] as input

Do you want to pursue the goal: _1674<_1675
   with the instantiated variables [] ?
|: y.
No clauses for _1674<_1675

When the goal: _2285 is invoked
 with the variables [] instantiated
which of the variables [_1674,_1675]
 become instantiated? (list of numbers)
|:
            .
            .
            .
```

At this point it is obvious where the problem lies - if it wasn't obvious when **split** was called with no arguments instantiated, it certainly is when an uninstantiated variable is posed as a goal.

The directed AND-OR tree search algorithm is given below. The more straightforward utility routines were omitted here for lack of space.

Algorithm 5: Iowalk

```
iowalk
iowalk(I,Goals) :-
 iowalk(I,O,Goals).

iowalk(I,O,(Goal,MoreG)) :- !,
 iowalk(I,O1,Goal),
 iowalk(O1,O,MoreG).

iowalk(I,O,Goal) :- !,
 nl,writel(['Do you want
              to pursue the goal: ',
            Goal]),
 varsof(Goal,I1),
 strict_intersect(I,I1,I2),
 nl,tab(2),
 writel([' with the instantiated
           variables ',I2,' ?']),
 nl,read(A),
 iorecurse((I,O,Goal),A).

iorecurse
iorecurse((I,O,Goal),n) :- !,
  ioquery(I,O1,Goal),append(O1,I,O).

iorecurse((I,O,Goal),y) :-
 clauses_of((I,Goal),L),!,
 pprels(L),
 setof(O1,
       (Goal1,I1,L)^
         (member((I1,Goal1),L),
          iowalk(I1,O1,Goal1)),
       Olist),
 collectO(Olist,List),
 append(I,List,O).

iorecurse((I,O,Goal),y) :-
 writel(['No clauses for ',Goal]),
 ioquery(I,O1,Goal),append(O1,I,O).

ioquery(I,O,Goal) :-
 varsof(Goal,V),
 strict_diff(V,I,U),
 ioquery1(I,U,O).

ioquery1(I,[],[]) :- !.
ioquery1(I,U,O) :-
 nl,writel(['When the goal: ',Goal]),
 nl,writel([' is invoked with the
          variables ',I,' instantiated']),
 nl, writel(['which of the variables ',
             U,' become instantiated?
                   (list of numbers)']),
 nl,read(N),
 setofO(X,Y^(member(Y,N),nth(U,Y,X)),O).
```

```
clauses_of((I,Goal),L) :-
 setof((I1,Goal1),
       (I,I2,Goal,V)^
            (clause(Goal,Goal1),
             varsof(Goal1,V),
             varsof(I,I2),
             strict_intersect(I2,V,I1)),
       L).

collect0(Olist,List) :-
 setof(V,L^(member(L,Olist),
            member(V,L)),List).
```

The **iowalk** sytem can be viewed as a symbolic trace package. Notice that beside representing the input and the output explicitly, this system also allows the user to omit any part of the tree he wants. This is not quite analogous to the "skip" of the usual Prolog debugging package[6] since there the execution is actual and is only invisible to the user. Here the execution is symbolic and the user may at any point supply information to influence the remainder of the traversal.

The reader may have noticed that **iowalk** repeats trying to pursue directed relations, although similar or identical ones have been encountered before. We in fact have another system called **iowalkc** which caches results as it goes along, and in the future tries to use those results before querying the user. Results can be cached either literally, or can be generalized. For example, after the predicate **write(hello)** is processed the user is given the option of generalizing the result, and in this example he would probably be wise to generalize the argument to **write**. This raises the issue whether we allow errors in the cached results. If we do, and our view at the moment is that we should, we have to allow for corrections of the cache when those errors manifest themselves. We envision a system in which directed relations will accumulate over a long time and across many users, so that it will be prohibitive to delete all the cached results and start from scratch. The alternative we suggest is a data-dependency system ([McDermott 83], [Doyle 79]) that will direct the undoing of wrong hypotheses selectively, and we have an initial implementation of such a data-dependency system. Since the scope of of these issues is beyond what we intended for this paper and because of the early stage of implementation we will not pursue this discussion any further here.

5 Related work

In 1957 McCarthy addressed the problem of inverting recursive functions [McCarthy 56], pointing out the difficulty of the problem.[7] The one method he discussed explicitely is the enumeration procedure, which is the analog of proving a theorem by systematically generating English text and testing to see if the text is a correct proof of the theorem. He speculated on what would be needed to improve upon this procedure, and one can consider the work described here a continuation of those speculations.

More recently Dijkstra has also considered the problem of program inversion. In [Dijkstra 83] he gives a (manual) inversion of the vector inversion problem. As he himself says, that inversion is straightforward because "the algorithm is deterministic and no information is lost", while the general inversion problem remains open.

In an interesting paper Toffoli ([Toffoli 80]) suggests a way of transforming any computational circuit to an equivalent invertible one with a worst case additional cost of doubling the number of channels. While the scope of this paper does not permit a detailed discussion of his work, there are two basic ideas - add "redundant" information to ensure function inversion, and try to reduce entropy by making the redundent information to one function be essential information for another function. Other references to theoretical work on reversible computations are [Bennett 73], [Burks 71], [Toffoli 77].

[Sickel 79] is work on invertibility in the context of logic programming. The notion of "j-invertibility" developed there is different from our notion of inversion. First, in [Sickel 79] a predicate which is a function from S1 to S2 is j-invertible if its direction includes the tuple $<S1 \cup S2-\{V_j\},\{V_j\}>$ where V_j is the j'th argument of the predicate. More importantly Sickel does not refer to the particular traversal order of Prolog, and programs that she considers "invertible" would in fact fail in Prolog. She gives two algorithms for determining the "input-output mapping" of a given predicate which are similar to our **iowalk** system.

6 Summary

- We suggest viewing Prolog predicates as denoting directed relations. For a predicate denoting a relation with a certain direction, we asked whether its direction can be extended. A major part of the paper has been concerned with the special case of function inversion.

[6] cf. "Prolog debugging Facilities" by Lawrence Byrd, in the documentation for Prolog-10.

[7] We do not agree with his claim there that solving any "well specified" problem amounted to the inversion of some Turing Machine. In our notation a specification procedure is a $\{<S,[]>\}$-directed relation R (i.e. a function) for some S and R, while the algorithm solving it is not the $\{<[],S>\}$-directed R but rather the $\{<S1,S2>\}$-directed R for some partition $<S1,S2>$ of S. This however does not affect the relevance of his subsequent discussion of inverting functions defined by Turing Machines.

- We have presented essentially two effective algorithms for inverting functions - Algorithm 2 and Algorithm 3. Both involve reversing the bodies of encountered clauses, but the latter is more selective in which clauses are inverted. Both allow for extra-logical features of Prolog, namely inverting assert/retract and arithmetic operations. The treatment of the latter is very cursory and ad-hoc, and if any non-trivial inversion of mathematical functions is desired the question of the representation of mathematical objects requires closer attention.
- It has been demonstrated that these algorithms are effective in some non-trivial cases, and that there exist functions not invertible by either. We have given some intuitive characterization of the functions invertible by each algorithm; the next step should be to make this characterization formal.
- A complete yet impractical algorithm for predicate redirection has been presented (namely a breadth-first search of the computation tree) and its performance has been demonstrated.
- The cut sign should be used judiciously in a function that is to be inverted, and conversely we should handle it more carefully in our algorithms.
- We presented a system that helps the user find out the consequences viewing a predicate as a relation with a certain direction.

Acknowledgements

Thanks go to various members of the FORBIN project for useful comments, especially to Tom Dean, Stan Letovsky and Jim Spohrer.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defence and monitored under the Office of Naval Research under contract N00014-83-K-0281.

References

[Apt & Van Emden 82] Apt, K.R. and Van Emden, M.H. Contributions to the Theory of Logic Programming. *JACM* 29(3), 1982.

[Bennett 73] Bennett, C.H. Logical Reversibility of Computation. *IBM J. Res. Dev.* 6, 1973.

[Burks 71] Burks, A.W. *On Backwards-Deterministic, Erasable, and Garden-of-Eden Automata.* Technical Report 012520-4-T, Comp. Comm. Sci. Dept., University of Michigan, 1971.

[Clocksin & Mellish 81] Clocksin, W.F. and Mellish, C.S. *Programming in Prolog.* Springer-Verlag, 1981.

[Dijkstra 83] Dijkstra, E.W. *EWD671: Program Inversion.* Springer-Verlag, 1983, .

[Doyle 79] Doyle, J. A Truth Maintenance System. *Artificial Intelligence* 12, 1979.

[Jones & Mycroft 83] Jones, N.D. and Mycroft, A. Stepwise Developement of Operational and Denotational Sematics for Prolog. In *Proc. of the First International Conf. on Logic Programming.* IEEE, Atlantic City, N.J., 1983.

[Shoham 84] Shoham, Y. FAME: A Prolog Program That Solves Problems in Combinatorics. In *Proc. 2nd Intl. Logic Programming Conf.*. Uppsala, Sweden, 1984.

[Sickel 79] Sickel, S. Invertibility of Logic Programs. In *Proc. Fourth Workshop on Automated Deduction.* Austin, Texas, 1979.

[Toffoli 77] Toffoli, T. Computation and Construction Universality of Reversible Cellular Automata. *J. Comp. Sys. Sci.* 15, 1977.

[Toffoli 80] Toffoli, T.
Reversible Computing.
Technical Report MIT/LCS/TM-151, Laboratory for Computer Science, MIT, February, 1980.

[Van Emden & Kowalski 76] Van Emden, M.H. and Kowalski, R.
The Semantics of Logic as a Programming Language.
JACM 23(4), October, 1976.

[Kowalski 74] Kowalski, Robert.
Predicate Logic as a Programming Language.
In *Proc. IFIP congress.* North-Holland, Stockholm, 1974.

[Lenat 82] Lenat, D.B.
AM: Discovery in Mathematics as Heuristic Search.
In Davis, R. and Lenat, D.B (editor), *Knowledge-Based Systems in Artificial Intelligence,* . McGraw-Hill, 1982.

[McAllester 83] McAllester, D.A.
The Theory and Practice of Mathematical Ontology: A Proposal.
Artificial Intelligence Laboratory, MIT , 1983.
unpublished at this time.

[McCarthy 56] McCarthy, J.
The Inversion of Functions Defined by Turing Machines.
In Shannon, C.E. and McCarthy, J. (editor), *Automata Studies,* . Princeton University Press, 1956.

[McDermott 83] McDermott, D.V.
Contexts and Data Dependencies: A Synthesis.
IEEE Transactions on Pattern Analysis and Machine Intelligence 5, 1983.

[Robinson 65] Robinson, J.A.
A Machine Oriented Logic Based on the Resolution Principle.
JACM 12(1), 1965.