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 computation tree, which is a computation tree in which 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 Emde Boas & Kowalski 78] 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 & Mcycroft 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 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 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:

\[
\text{qsort}(\text{H}([T], \text{S})), \text{L}.
\]

\[
\text{split}(\text{H}, \text{T}, \text{A}, \text{E}) \leftarrow
\text{qsort}(\text{A}, \text{A}),
\text{qsort}(\text{B}, \text{B}),
\text{append}(\text{A}, \text{H}([\text{B}], \text{S})),
\text{qsort}(\text{L}, []).
\]

\[
\text{split}(\text{H}, \text{A}[\text{X}], \text{A}[\text{Y}], \text{Z}) \leftarrow
\text{order}(\text{A}, \text{E}), \text{split}(\text{H}, \text{X}, \text{Y}, \text{Z}).
\]

\[
\text{split}(\text{H}, \text{A}[\text{X}], \text{Y}, \text{A}[\text{Z}]) \leftarrow
\text{not}(\text{order}(\text{A}, \text{E})), \text{split}(\text{H}, \text{X}, \text{Y}, \text{Z}).
\]

\[
\text{order}(\text{A}, \text{E}) \leftarrow \text{A} \leq \text{B}.
\]

One would expect invocation of the goal

\[
\text{qsort}(\text{X}, [1, 2, 3])
\]

to bind \text{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 Insertion sort:

\[
\text{insert}([], []).
\]

\[
\text{insert}([\text{X}], [\text{Y}]) \leftarrow
\text{insert}([\text{X}, \text{Y}]), \text{insert}(\text{X}, \text{X}, \text{Y}, \text{Y}).
\]

\[
\text{insert}([\text{X} \leq \text{Y}], [\text{X}, \text{Y}, \text{Z}]) \leftarrow
\text{not}(\text{order}(\text{X}, \text{Y})),
\text{insert}([\text{X}, \text{Y}, \text{Z}]),
\text{order}(\text{X}, \text{Y})
\]

\[
\text{order}(\text{A}, \text{E}) \leftarrow \text{A} \leq \text{B}.
\]
insert(X, [], [X]).
insert(X, [X|L], L) :-
order(X, A), !.
insert(X, [A|L], L) :-
insert(X, L, N).

where order is defined as above. When the goal
insert(X, [1,2,3], 1) 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 *modes* to use Prolog-10 terminology. In
this case we might say that sortedaa(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 \(R[x_1, \ldots, x_n]\) be an n-
ary Prolog predicate with an intended
interpretation \(I =\langle x_1^i, \ldots, x_n^i \rangle\), and let
\(V =\langle x_1, \ldots, x_n \rangle\). It is said to be a function
from \(V_1\) to \(V_2\) if \(\langle V_1, V_2 \rangle\) is a
partition of \(V\); and for all instantiations of \(V\)
and thus partial instantiations of \(V_1\) and \(V_2\),
invoking \(R\) will fairly generate all the
resulting \(\langle V_1, V_2 \rangle\) tuples.

For our purposes a partition of a set \(S\) is
two 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 \([\{\}\] to \([X]\).

**Definition:** A Prolog predicate \(R\) with
a given intended interpretation is said to be
\(D\)-directed relation if \(D\) is a set of
tuples \(\{\langle V_1, V_2 \rangle\}\) such that \(R\) is a
function from \(V_1\) to \(V_2\) for all \(i\). Note
that a function from \(V_1\) to \(V_2\) is a special
case of a directed relation, one that is
\(\{\langle V_1, V_2 \rangle\}\)-directed.

**Example:** qsort(X,Y) is
\(\{\langle X, Y \rangle, \{\langle X, Y \rangle \}\}\)-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 \(R\).

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 \(V_1\) to \(V_2\), and
we want to extend it to be
\(\{\langle V_1, V_2 \rangle, \langle V_2, V_1 \rangle\}\)-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
cactoring 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], [Lonat 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 conjunction in reverse order. Given a
single goal reduce it if possible, otherwise execute it*.

**Algorithm 1: A simple inversion**

\[\text{invgoal}(A,B) :-\]
\[\text{1. invgoal}(B), \text{invgoal}(A).\]
\[\text{invgoal}(A) :-\]
\[\text{clause}(A,B), \text{invgoal}(B).\]
\[\text{invgoal}(A) :-\]
\[\text{not(clause} (A_1).), \text{call}(A).\]

When we apply the above algorithm to the sorting
programs from section 2 we observe the following
behavior:\(^3\)

---

\(^2\)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 nondeterministic function.

\(^3\)All the examples in this paper were done on a DEC20
running Prolog-10 version 3.47.
Example 1: Inverting QuickSort and Insert

?– 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(insert(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 86]), 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 invent 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 occurring 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
% inverses 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 1) for proving combinatorial equalities by combinatorial arguments. The general structure of proving an expression equal to 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 ! from set2?"

\[ \text{Solution} = c(r,1) \times c(n,r) \]

\[ \text{yes} \]

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

**Example 2:** inverting Count

\[ \text{X} = \{ (241, r), (368, 1), (242, a) \} \]

\[ \text{Y} = \{ \text{subset}(241, 242), \text{subset}(368, 241) \} \]

\[ \text{yes} \]

\[ \text{?} \]

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:

\[ \text{abs}(N, M) \leftarrow \text{not}(N > M). \]

Given the goal \(\text{abs}(X, 2)\) Algorithm 2 will execute as follows:

**Example 3:** Inverting \(\text{abs}\)

\[ \text{X} = -2 ; \]

**Error:** evaluate(_31)

\[ \text{X} = 2 ; \]

\[ \text{no} \]

\[ \text{?} \]

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 !.

\[ \text{?} \sim \text{invgoal}(\text{abs}(X, -2)). \]

**Error:** evaluate(_31)

\[ \text{X} = -2 ; \]

\[ \text{no} \]

\[ \text{?} \]

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:

\[ \text{abs}(N, M) \leftarrow \text{not}(N > M). \]

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 \(\text{F}\) is called \(\text{inv}(\text{F})\). 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 \(\text{inv}(\text{A}) :- \text{inv}(\text{B})\) and recursing on \(\text{B}\). 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 disadvantage 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 \(\text{gensym}\) 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 \(\text{gensym}(\text{"string",S})\) 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), !.
findinv1([A|B], X, Y) :-
  !, findinv(A, X, Y), findinv1(B, X, Y).
findinv1(_ , _, _) :- !.

findinv1(A, B, X, Y) :-
  !, findinv(A, X, Y), findinv1(B, X, Y).
findinv1(X, Y, Y) :-
  telling(F), tell(user), nl,
  write('Do you want to invert the goal ', A, '? (y/n) '), nl, !.
  read(A), !.
findinv1(X, Y, 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), findinv(C, X, Y).

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

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

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

invwrite(A, C, no) :- !.
  invwrite(A, C, yes) :- nl,
  write('Inv(A, C) :- inv(A,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(62, 219) ? (y/n) | y.
Do you want to invert the goal append(217, 216, 219) ? (y/n) | y.
Do you want to invert the goal true ? (y/n) | n.
Do you want to invert the goal append(515, 216, 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, [1, 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 _216<10 ? (y/n) | n.
Do you want to invert the goal integername(627, [629], 216) ?
(y/n) | y.
Do you want to invert the goal _626 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 getzumsym(31, 216) ? (y/n) | y.
Do you want to invert the goal asserta(gensymum(31, 216)) ?
(y/n) | n.
Do you want to invert the goal _216 is _219+1 ? (y/n) | n.
Do you want to invert the goal _216 is _219 ? (y/n) | n.
Do you want to invert the goal retract(gensymum(31, 219)) ?
(y/n) | n.
Do you want to invert the goal asserta(gensymum(31, 219)) ?
(y/a) | x. 
X = .31; 
Y = .62 

| ?- inv(genSym(X, input7)). 
X = input 

| ?- 
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:

\[ f([a,X]) : = g(X). \]
\[ f([b,X]) : = g(X). \]
\[ g([c,-]) : = f(X). \]

Considered as a function from \([X] \) to \([X] \), \(f([X]) \) acts as recognizing the regular language \((a+b)^+c.\Delta.\) 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-solutionness" (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 \((a+b)^+c.\Delta.\)

| ?- bf(f([X])). 
X = [a,c,1312]; 

\(^4\)The exact definition of \(bf \) is omitted for lack of space

| X = [b,c,1516]; 
X = [a,a,c,1342]; 
X = [a,b,c,1518]; 

| X = [b,b,a,a,c,14805]; 
X = [b,b,b,a,b,c,15399]; 
X = [b,b,b,a,c,15041]; 
! more core needed 
[ Execution aborted ] 
| ?- 

We actually have another implementation of \(bf \) that, using side effects, uses space more economically. The program for the \(bf \) is 30 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 \(<X,Y,Z],[A,B,C]>\). 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 \(S1=5=0\) and \(S2=2=0\) then \(S3=0\) ...). 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 instantiations 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. \(sort(\{1,0\},G)\)\(^5\) will initiate a preorder depth first search of the directed computation tree whose root is \(<1,0,G>\).

\(^5\)Here we borrow from the terminology of data dependency [McDermott 83]
Example 7: Walking the directed tree for Quicksort

| ?- icwalk([Y], qsort(X,Y)).
Do you want to pursue the goal:
qsort(52, 29)
with the instantiated variables [ ]?
| y.

The 10 relations in its subgoals are
The goal(s): true with [] as input
The goal(s):
split(480, 481, 482, 483), !,
qusort(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 [ ]?
| !.

The 10 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 [ ]?
| !.

The 10 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 [ ]?
| !.
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: icwalk

icwalk(I, Goals) :-
icwalk(I, Goals).
icwalk(I, (Goal, MoreG)) :- !,
icwalk(I, Goal).
icwalk(I, MoreG).
icwalk(I, Goal) :- !,
nl, write1(’Do you want to pursue the goal: ’, Goal),
varsof(Goal,I),
strict_intersect(I, I1, I2),
sl, tab(2),
write1(’ with the instantiated variables ’, I1, ’,’ I2, ’’)?’),
sl, read(A),
iorecurse(I, 0, Goal, A).
iorecurse
iorecurse(I, 0, Goal), n :- !,
iquery(I, 0, Goal), append(0I, I, 0).
iorecurse((I, 0, Goal), y) :-
clause_of((I, Goal), L),
pptels(G),
setof(0I,
    (Goal, I, L)
    (member(I, Goal), L),
icwalk(II, 0, Goal)),
Olist),
collectO(Olist, List),
append(I, List, 0).
iorecurse((I, 0, Goal), y) :-
write1(’No clauses for ’, Goal),
iquery(I, 0, Goal), append(0I, I, 0).
iquery(I, 0, Goal) :-
varsof(Goal,Y),
strict_diff(V, I, U),
iquery(I, U, 0).
iquery(I, [ , ]) :- !,
iquery(I, [ , U, 0] :-
nl, write1(’When the goal: ’, Goal),
nl, write1(’ is invoked with the variables ’, I, ’, instantiated’),
nl, write1(’which of the variables ’, U, ’, become instantiated?’
    (list of numbers)’),
nl, read(N),
setof(X, Y, (member(Y, N), nth(U, Y, X)), 0).
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 these 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 redundant 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.S2-(Vj),(Vj)> where Vj 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.

### 8 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.

---

5 Related work

In 1957 McCarthy addressed the problem of inverting recursive functions [McCarthy 58], pointing out the difficulty of the problem. The one method he discussed explicitly is the enumeration procedure, which is the analog of

---


---

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 \(<S||R>\)-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 \(<S|S2>\)-directed R for some partition \(<S|S2>\) 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 3 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, Sias Leivosky 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

[Ap't & Van Emden 82]
Apt, K.R. and Van Emden, M.H.
Contributions to the Theory of Logic Programming.

[Dennet 73]
Dennett, C.H.
Logical Reversibility of Computation.

[Burks 71]
Burks, A.W.
_On Backwards-Deterministic, Erasable, and Garden-of-Eden Automata._
Technical Report 012560-4-T,
University of Michigan, 1971.

[Clocks in & 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 Development of Operational and Denotational Semantics for Prolog.
_In Proc. of the First International Conf. on Logic Programming._

[Shoham 84]
Shoham, Y.
FAME: A Prolog Program That Solves Problems in Combinatorics.

[Sickel 79]
Sickel, S.
Invertibility of Logic Programs.

[Toffoli 77]
Toffoli, T.
Computation and Construction Universality of Reversible Cellular Automata.
[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.

Predicate Logic as a Programming Language.

[Lenat 82] Lenat, D.B.
AM: Discovery in Mathematics as Heuristic Search.

[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.

[McDermott 83] McDermott, D.V.
Contexts and Data Dependencies: A Synthesis.

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