Knowledge Inversion

Yoav Shoham

Drew V. McDermott

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

Abstract

We define the *direction* of knowledge, and what it means to extend that direction. A special case is function inversion, and we give three algorithms for function inversion. Their performance on non-trivial problems and their shortcomings are demonstrated. All algorithms are implemented in Prolog.

1 Introduction.

Given a manual describing how to assemble a machine, we can usually use that manual to disassemble the same machine; given our knowledge of differentiation of algebraic functions, we can integrate a variety of functions. On the other hand while it is trivial to disarrange Rubik's Cube it is less trivial to arrange it, as many have discovered to their frustration. We can then ask ourselves two questions:

- Can we characterize the instances of "easily invertible" knowledge?
- Can we automate the inversion of procedural knowledge in those easy cases?

In this paper we mainly ignore the first question, but give a partial positive answer to the second one. We present essentially three different algorithms for function inversion and demonstrate their power and weaknesses.

Our algorithms are implemented in Prolog ([Clocksin & Mellish 81]), which may seem at first a bit strange since the popular view of Prolog is as a "declarative" language. In section 2 we dispel this optical illusion which oddly enough is sometimes encouraged by the logic programming community itself. Our algorithms could be written in any applicative language that employs backtracking; Prolog happens to be particularly convenient because of the explicit representation of the output variables (or perhaps this is a post-hoc rationalization by the first author of his enthusiasm for the language - the reader may be the judge of that). We do not rely on the formalism of logic programming, but the reader is expected to have a basic understanding of deductive systems like Prolog or DUCK ([McDermott 82]) and of the syntax of Prolog.

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

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]) :-
 order(H,A), 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. Other cases are still worse - replacing Quicksort by Insertionsort will cause the interpreter to go into an infinite recursion, and similar disasters will happen with Bubblesort.

The problem is obviously that goals are invoked with the "wrong" arguments instantiated. 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: A Prolog predicate R with a given intended extension is said to be a <u>function from S1 to S2</u> if <S1,S2> is a partition of the set of all variables appearing in R, and for all invocations of R with all the variables in S1 instantiated, all the tuples in the intended extension of R matching the instantiation of variables in S1 will be fairly generated.

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.

Definition: A Prolog predicate R with a given intended extension is said to be <u>D-directed</u> relation if D is a set of tuples {<S1i,S2i>} such that R is a function from S1i to S2i for all i. Note that a function from S1 to S2 is a special case of a directed relation, one that is {<S1,S2>}-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 a given predicate in a given 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 S1 to S2, and we want to extend it to be {<S1,S2>,<S2,S1>}-directed, that is we want to invert the function. In another paper ([Shoham & McDermott 84]) we describe a general procedure for exploring a directionality of a given predicate in a given program. Here we restrict the discussion to function inversion, which is the subject of the next section.

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. Remember however that we are not trying to invert *all* functions, but rather are investigating which ones are easily invertible. Thus the algorithms we present are really heuristics for function inversion. In this section we are concerned with a detailed description of the algorithms and their performance; we return to more global considerations in section 4.

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". The precise Prolog implementation is given in Figure 1.

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

Example 1: inverting Quicksort

```
| ?- 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
| ?-
```

[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.

which is indeed what is required. However this inversion procedure is too simplistic as it does not take into account some of Prolog's idiosyncracies. In Figure 2 we present a procedure that adopts the same basic algorithm, but pays more respect to special Prolog features.

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 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.

Figure 3 is an example of a counting problem solved 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**?").

We now ask the converse question - "What counting problem is the expression `c(n,r)*c(r,1)` a solution to" by inverting `count`. The result is shown in figure 4.

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

```
invgoal((A,B)) :- !,invgoal(B),invgoal(A).
invgoal(A) :- !,clause(A,B),invgoal(B).
invgoal(A) :- call(A).
```

Figure 1: Algorithm 1: A simple inversion

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

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

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

invgoal((A,B)) :- !,invgoal(B),invgoal(A).
invgoal(A) :- !,clause(A,B),invgoal(B).
invgoal(A) :- call(A).
```

Figure 2: Algorithm 2: A less simple inversion

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

Solution = c(r,1)*c(n,r)

yes
| ?-
```

Figure 3: Solving a counting problem

```
| ?- 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
| ?-
```

Figure 4: Example 2: inverting Count

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 computation tree and whenever a goal 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`.[4] 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. The disadvatage is that when the user decides to quit pursuing a branch of the tree he may lose information. The example we choose 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.

```
| ?- 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.


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

X = input

yes
| ?-
```

Figure 5: Example 5: inverting gensym

Finally, we demonstrate that 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).
```

[4]The limitations imposed on the length of this presentation prohibit a more detailed description of the algorithm or a complete I/O log.

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.

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.

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

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

X = [a,c,_312] ;
X = [b,c,_516] ;
X = [a,a,c,_1342] ;
X = [a,b,c,_1618] ;
         .
         .
         .
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 ]
```

4 Related work, Summary, Further Research.

4.1 Discussion of related work.

In 1956 McCarthy addressed the problem of inverting recursive functions [McCarthy 56], pointing out the difficulty of the problem.[5] 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.

[5]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.

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 insure function inversion, and try to reduce entropy by making the redundant information to one function be essential information for another function. The motivation behind that work is different from ours, but we feel that the two basic ideas may carry over (see last subsection). Other references to theoretical work on reversible computations are [Bennett 73], [Burks 71], [Toffoli 77].

4.2 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.
- We have presented 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. The exact characterization of functions invertible by each algorithm has not been given (see discussion below).
- 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.

4.3 Further research.

We repeat the two questions posed in the introduction:

- Can we characterize the instances of "easily invertible" knowledge?
- Can we automate the inversion of procedural knowledge in those easy cases?

As we said there, we only gave a partial answer to the second question. The task remains, then, to complete that answer and provide one for the first question. Several ways of approaching the first half of the task suggest themselves. First, we have not explored the power of combining techniques - for example perform Algorithm 2 and add the clause `invgoal(bf(G)) :- bf(G).` A related issue that needs exploring is how to make all implicit knowledge explicit. For example, if we write the definite clause `f(X,X,Z)` with the intention that f(X,Y,Z) be used as a function from [X,Y] to [Z], we sometimes ignore additional (overdetermining) information, for example that if X=Y then Z=[]; this sort of information may be crucial for the inversion of `f`.

While we have mentioned and discredited BFS as a sole strategy for searching the computation tree, we have not mentioned other possible strategies. One obvious candidate is a probablistic one - the interpreter could flip a coin to decide on which clause to resolve against the current goal, and even to decide on the ordering of a clause body. Another approach could be more in the spirit of mainstream AI, that the choice of ordering itself be a knowledge-intensive problem solving task. A recent paper ([Smith & Genesereth 83]) has concerned itself with part of the problem, that of deciding on the optimal ordering of conjuncts in the simple case where those conjuncts resolve only against unqualified assertions in the data base (that is, no further inference is necessary).

Finally, Toffoli's work suggests both an approach for answering the first question as well a technique answering the second one. It implies that one should look for an appropriate measure of entropy in the computation, and try to minimize it. In the case where the entropy is zero the computation is invertible. Where the we cannot eliminate the loss of information, we should try to supply excess information at the start, so that we could reconstruct just the "right" subset of it later. This also suggests application of the automatic programming paradigm, whereby the process of adding redundant information to an existing piece of code is automated.

We have allowed ourselves some free speculation in this last subsection, which reflects our excitement with the possibilities. It is not clear why the problem has been largely neglected - for example in the survey of machine learning ([Michalski et al 83]) there is no mention of knowledge inversion as part of skill acquisition. Whether knowledge inversion is classified as part of a learning process or not it seems a fundamental capability of people, and AI will benefit much from a better understanding of it.

ACKNOWLEDGMENTS

Thanks go to Tom Dean, Stan Letovsky, Dave Miller and Jim Spohrer for helpful comments on previous drafts. We also thank one referee for correctly pointing out related work by S. Sickel of which we had not been aware.

References

[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, .

[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 82] McDermott, D.V.
DUCK: A Lisp-Based Deductive System.
Yale University, Department of Computer Science , 1982.

[Michalski et al 83]
Michalski, R.S., Carbonell, J.G. and Mitchell, T.M.
Machine Learning: An Artificial Intelligence approach.
Tioga, 1983.

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

[Shoham & McDermott 84]
Shoham,Y. and McDermott, D.V.
Prolog Predicates as Denoting Directed Relations.
submitted , 1984.

[Smith & Genesereth 83]
Smith, D.E. and Genesereth, M.R.
Ordering Conjuncts in Problem Solving.
Computer Science Department, Stanford University , 1983.
unpublished at this time.

[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.