A Heuristic Approach to Program Inversion

David Eppstein

Computer Science Department
Columbia University
New York, N.Y. 10027

Abstract

A notation is given for describing the inverse of multiple functions and of functions of multiple arguments. A technique based upon this notation is presented for taking a program written in pure LISP and automatically deriving a program which computes the inverse function of the given program. This technique differs from previous such methods in its use of heuristics to invert conditionals.

1. Introduction

There are many applications in which it is useful to compute the inverse of some program, that is, to find another program such that feeding the output of the original program as input to the new program produces the original input. One such application is in programming by specification: one would like to define a program to compute the square root of a number by the equation $(\sqrt{x})^2 = x$ rather than supplying an actual iterative method of solving the equation. Another application of program inversion is in debugging. Given a program and an erroneous result, one would like to step backwards through the program to trace the source of the error. Yet a third application is in transforming the input domain of a program, such as the well known technique of multiplying polynomials by first performing an FFT; to get the result back into the original domain one needs to find an inverse transformation.

Several methods have been suggested for performing such inversion. Unfortunately they each suffer from several defects. McCarthy [3] suggests a generate and test approach; this will correctly find an inverse when it exists, but is computationally infeasible and cannot determine whether the inverse is unique. Dijkstra [1] provides a technique for inverting programs symbolically, but requires that the programmer provide inductive assertions on conditional and loop statements. Korf [2] suggests another method that automatically provides these assertions, but recursions derived using his method are not guaranteed to be well founded. Several recent efforts [5,6] have gone into inverting Prolog; this differs from inverting other programming languages in that Prolog is less procedural and more declarative. Methods for inverting procedural languages will thus also be useful for Prolog, but the reverse is not necessarily true.

This paper suggests a method of providing these assertions automatically by heuristic methods. This method will not always find an inverse for a program, but when one is found it will always correctly terminate when the output of the original program is given as input. In addition, if an inverse is found it will be the case that the original function was one-to-one—in principle the inverse could itself be inverted to recover the original program.

The method described here has been implemented as a MacLISP program. The program was able to derive the inverse of `append` (shown in detail below), a version of `reverse` as given in [2], unary integer negation defined recursively using `add1` and `sub1`, and several other such small programs. The example of unary negation is especially interesting as a case where Korf's method is unable to find the inverse, because of the existence of two recursive clauses in its definition.

2. Inversion of Multiple Functions of Multiple Arguments

The usual definition of the inverse of a function $f\colon D \to R$ is the function $f^{-1}\colon R \to D$ such that $f^{-1}(f(x)) = x$ for all x in D. Unfortunately this notation does not lend itself well to inversion of functions of more than one argument. For instance, in the LISP programming language it is true that (`car` (`cons` x y)) $= x$ and (`cdr` (`cons` x y)) $= y$. Thus one intuitively thinks of `car` and `cdr` together as being the inverses of `cons`, but there is no one function that we can call $\texttt{cons}^{-1}$.

On the other hand, we can express a relation between f and f^{-1} that does extend to multiple arguments and also to multiple functions: If we assume that $y = f(x)$, then by the above definition $f^{-1}(y) = x$, and the converse similarly holds. Thus we have

$$y = f(x) \quad \text{if and only if} \quad f^{-1}(y) = x.$$

Similarly,

$$z = (\texttt{cons}\ \ x\ \ y)$$

is true if and only if

$$(\texttt{car}\ \ z) = x$$

and

$$(\texttt{cdr}\ \ z) = y$$

are both true. Taking this as our definition of inversion, we say that `cons` inverts to `car` and `cdr`.

In general we will say that some list of n functions fun_i of m arguments each inverts to another list of m functions inv_j of n arguments each if, for all x_j and y_i, the set of equations

$$\begin{aligned} y_1 &= \mathit{fun}_1(x_1, x_2, \ldots, x_m) \\ y_2 &= \mathit{fun}_2(x_1, x_2, \ldots, x_m) \\ &\vdots \\ y_n &= \mathit{fun}_n(x_1, x_2, \ldots, x_m) \end{aligned}$$

simultaneously hold if and only if the inverse set of equations

$$\begin{aligned} \mathit{inv}_1(y_1, y_2, \ldots, y_n) &= x_1 \\ \mathit{inv}_2(y_1, y_2, \ldots, y_n) &= x_2 \\ &\vdots \\ \mathit{inv}_m(y_1, y_2, \ldots, y_n) &= x_m \end{aligned}$$

also simultaneously hold. Note that this relationship is symmetric: inverting the inverses of a list of functions produces the original list. The requirement that all functions have the same number of arguments turns out not to be a problem; if necessary we can add dummy arguments to those functions that need them.

Since we are attempting to invert programs, we will need to have inverses for the primitive operations of the language we are inverting. In LISP, we will use the fact that `cons` inverts to `car` and `cdr`, and that `add1` inverts to `sub1`. These can be used to define the other arithmetic and list manipulation functions usually found in a LISP implementation.

3. The Inversion Method

We perform inversion as a search through a state space of program descriptions. Each state is a set of facts, each of which is composed of a left side, a right side, and a set of preconditions. A fact can be interpreted as meaning that, if the preconditions all hold, the left side will be equal to the right side. We will use four operators to move from state to state: *conditional expansion*, *precondition replacement*, *expression inversion*, and *conditional contraction*.

For example, suppose we want to invert the function `append`. That is, we want to find two programs that, given the result of a call to `append`, will return the first and second arguments. Unfortunately there are many possible pairs of arguments that could have produced the same result from `append`; thus we need to introduce an auxiliary function to distinguish among them. One function we might use returns the length of its first argument and ignores its second; we will call it `flength`. We will call the arguments to it and `append` by the names `firstn` and `lastbutn` for reasons that we shall see below. The definitions of our two original functions give us our initial state:

```
append = (cond ((null firstn) lastbutn)
               (t (cons (car firstn)
                        (append (cdr firstn)
                                lastbutn))))
flength = (cond ((null firstn) 0)
                (t (add1 (flength (cdr firstn)
                                  lastbutn))))
```

The atoms on the left sides correspond to a call to each of the functions being defined, and those on the right side correspond to the arguments to those calls; there are no unbound variables. All initial facts have no preconditions, and their left sides are all atomic. Similarly we will define a goal state as one in which all sets of preconditions are empty and the right sides are all atomic. No new atoms will be introduced by our transformations, and each atom will occur only on one side of our facts; thus the original arguments will become the names of the inverse programs, and the original program names will become the inverse arguments.

Both facts in the current state have no preconditions, but the right side of each is a conditional expression. Before we can perform any other transformations on the facts, we first separate out the conditional parts into preconditions. The conditional expansion operator does this; it replaces a fact for which the right side is a conditional expression with two facts, one corresponding to the case when the predicate in the condition is true and one corresponding to the case when it is false. Applying this to each of the facts in our initial state produces a new state:

```
(null firstn)
   → append = lastbutn
   → flength = 0
(not (null firstn))
   → append = (cons (car firstn
                     (append
   → flength = (add1 (flength
```

Note that the above state consists of four facts, each of which has a precondition of either (`null firstn`) or (`not` (`null firstn`)). The preconditions are combined in the above display merely for the sake of brevity.

In our final inverse programs, we will need a conditional expression to determine which path the original programs took. Another way to think of this is that the preconditions in the current state are functions of the arguments to the original functions, and we would like to replace them with new functions of the arguments to the inverse functions that are true exactly when the old ones were. This replacement is the heuristic portion of the inversion; in general there will be many possible preconditions but there seems to be no analytic method of finding them.

The details of how we find the new precondition will be described below; in this case we notice that `flength` will be zero if and only if `firstn` is null. Once we have found our new precondition, we can use the precondition replacement operator. Note that if we simply replaced all occurrences of the old precondition with the new one we would derive the useless fact

```
(zerop flength) → flength = 0
```

but we would not produce

```
(zerop flength) → nil = firstn,
```

without which we could not complete our derivation. Taking this into account, we come up with a new transformed state:

```
(zerop flength)
   → append = lastbutn
   → nil = firstn
(not (zerop flength))
   → append = (cons (car firstn)
                    (append (cdr firstn) lastbutn))
   → flength = (add1 (flength (cdr firstn) lastbutn))
```

Now the derivation goes through a sequence of expression inversions, pulling functions from the right side of facts over to the left. This process is completely mechanical, but it is somewhat complicated so we will go through it one step at a time. First we notice that we have a call to `add1` as the outer call of the right side of a fact. When we invert this call to `sub1`, our state becomes

```
(zerop flength)
   → append = lastbutn
   → nil = firstn
(not (zerop flength))
   → append = (cons (car firstn)
                    (append (cdr firstn) lastbutn))
   → (sub1 flength) = (flength (cdr firstn) lastbutn)
```

In this case we inverted one function to one function, so the total number of facts in our state didn't change. When we invert `cons` to `car` and `cdr`, however, we get two facts where before we had one:

```
(zerop flength)
   → append = lastbutn
   → nil = firstn
(not (zerop flength))
   → (car append) = (car firstn)
   → (cdr append) = (append (cdr firstn) lastbutn)
   → (sub1 flength) = (flength (cdr firstn) lastbutn)
```

The above inversions were done on functions for which already knew the inverses. We must also invert recursive calls to the original functions into recursive calls to our new inverses. Since in this case we have two functions of two arguments each, we will replace two facts with two new facts. This inversion cannot be split into two steps, because each new fact has parts from both of the old facts. Note also that the arguments in the calls have to match exactly; it would not have changed the result of `flength` if the second argument in the recursive call were different, but it would have made it impossible to complete our inversion. Performing the inversion, we get to a new state:

```
(zerop flength)
   → append = lastbutn
   → nil = firstn
(not (zerop flength))
   → (car append) = (car firstn)
   → (firstn (cdr append)
             (sub1 flength)) = (cdr firstn)
   → (lastbutn (cdr append)
               (sub1 flength)) = lastbutn
```

Now all but two of the right sides are atomic; these two can be made atomic by one more inversion, from `car` and `cdr` into `cons`.

```
(zerop flength)
   → append = lastbutn
   → nil = firstn
(not (zerop flength))
   → (cons (car append)
           (firstn (cdr append)
                   (sub1 flength))) = firstn
   → (lastbutn (cdr append)
               (sub1 flength)) = lastbutn
```

We are almost at a goal state; the only remaining task is to remove the preconditions. This can be achieved by the conditional contraction operator, which acts inversely to conditional expansion: it takes two facts with opposite preconditions and identical right sides, and combines them into one fact having as a left side a conditional expression evaluating to either of the two previous left sides depending on which precondition is true. We must be careful here only to accept preconditions that are in terms of the inverse arguments, or we would not come up with a well-defined program; this is the reason for our precondition replacement above. Two applications of conditional contraction produce a goal state:

```
(cond ((zerop flength) nil)
      (t (cons (car append)
               (firstn (cdr append)
                       (sub1 flength))))) = firstn
(cond ((zerop flength) append)
      (t (lastbutn (cdr append)
                   (sub1 flength)))) = lastbutn
```

4. Precondition Replacement

The above method provides a framework for inversion; we also need heuristics to be used in that framework for finding replacement preconditions. One such heuristic that is effective for many simple recursions uses a sort of data type system. All objects are members of type *top*, which is divided up into integers, `cons`es, and `nil`. Unlike most data type systems we have types corresponding to major subsets of the integers: the negative numbers, the positive numbers, zero, and the unions of pairs of these sets. The main requirement on our type system is that it form a lattice; thus we also need a type *bottom* to which no object belongs.

Functions are assigned types that contain all their possible return values. Thus `cons` always returns a `cons`, and `add1` always returns a number, but `car` and `cdr` can return anything and so their type is top. The type of a function should itself be a function mapping from the function's input types to its output type, but for simplicity this is only actually done for `add1` and `sub1` (e.g. `add1` will return positive numbers if its argument is non-negative).

Given a list of new functions to be inverted, or a list of inverse functions to be used in further inversions, we can calculate their types by a simple relaxation process: we start by assuming that the return type of each of them is bottom. Then we use that in evaluating the types of the expressions defining them (making no assumptions about the types of their arguments) to arrive at a new assignment of types, and iterate until no function's type is changed by the iteration. With our definition of `flength`, for example, the first iteration would result in a type of zero from the base clause, and bottom from the recursive clause, which combine to a type of zero. Then the second iteration gives the same base clause type but the recursive clause is now known to return positive numbers, and so the new type becomes that of the non-negative numbers. Adding one to a non-negative number as with adding one to zero returns a positive number, so in the third iteration the overall type doesn't change. Thus the final type of `flength` is that of the non-negative numbers. In a more complicated recursion the preliminary types of the recursive clauses might cause their new types to change, and so there could be up to as many iterations as the depth of the type lattice.

Now we can use our type system to provide replacement preconditions. This is done by looking for two facts with identical left sides, and identical preconditions except for the particular precondition we wish to replace, which should be true for one fact and false for the other. Then we calculate the types of the right sides of the facts, and if the intersection of the two types is bottom then we can replace our precondition with a test for membership on the common left side in one of the two types. Thus in the `append` example,
for the fact with precondition `firstn` true, and it is positive (and therefore not zero) with `firstn` false.

5. Future Work

The main obstacle to inversions with the current implementation is the inability of the heuristic described above to find replacement preconditions that are functions of more than one argument. More effort could be put into methods of finding complex predicates; it might prove fruitful for heuristics to notice whether the function being inverted is arithmetic or list processing and tailor the search accordingly.

In the inversion of `append` we needed to introduce an auxiliary function. Because `append` recurs linearly we could simply count the number of recursive calls, but this will not always be sufficient. Work could be done in automatic detection of the need for such an auxiliary function, and in automatic generation of the function when its need is detected.

Some thought in the program transformation world has gone into the idea of a MACSYMA-like system for computer programs. The representations and operators used in this paper appear useful in other domains than inversion; one might study how applicable they would be in a more general program transformation system, and how they could be incorporated into such a system.

Acknowledgements

I would like to thank my advisor, Rich Korf, for many helpful discussions and suggestions. This research was sponsored in part by a National Science Foundation student fellowship and by the Defense Advanced Research Projects Agency under contract N00039–84–C–0165.

References

[1] Dijkstra, E. W., "Program inversion." In *Program Construction*. Springer-Verlag, Berlin, 1979.

[2] Korf, Richard E., "Inversion of applicative programs." In *Proc. IJCAI-81*. Vancouver, B.C., 1981.

[3] McCarthy, John, "The inversion of functions defined by Turing Machines." In Shannon, C.E., and J. McCarthy (editors), *Automata Studies*. Princeton University Press, Princeton, N.J., 1956.

[4] Scherlis, W. L., "Expression procedures and program derivation." Ph.D. thesis, Department of Computer Science, Stanford University, 1980.

[5] Shoham, Yoav, "Knowledge inversion." In *Proc. AAAI-84*. Austin, Texas, 1984.

[6] Sickel, Sharon, "Invertibility of Logic Programs." In *Proc. Fourth Workshop on Automatic Deduction*. Austin, Texas, 1979.