Inversion of Applicative Programs

Richard E. Korf

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

and

IBM Computer Sciences Department
Thomas J. Watson Research Center
Yorktown Heights, N.Y. 10598

Abstract

A technique is presented for taking a program written in pure LISP and automatically deriving a program which computes the inverse function of the given program The scheme is based on a set of rules for inverting the primitive functions plus a method of solving for variables introduced by non-invertible functions. As an example, a program to reverse a list is inverted.

1. Introduction

This paper describes ongoing research on the problem of automatically inverting applicative programs. The motivation for this work is to achieve a better understanding of the transformation of computer programs. Inversion is an important program manipulation problem. In addition, program inversion may prove to be a fruitful special case for studying the general transformation problem The domain of purely applicative programs is chosen because the absence of side effects results in much cleaner program manipulation.

The program inversion problem was originally addressed by McCarthy 13] in the context of Turing machines Sickel [6] provides an algorithm for determining the invertibility of logic programs. Dijkstra [2] presents a manual derivation of the inverse of an imperative program. A rule for inverting the cons construct in applicative programs is described by Scherhs [5]. Wadler [7] has implemented a limited inversion facility as pari of the interpreter for his *What?* language. Darlington's [1] program transformation system handles inversion by the folding and unfolding of recursion equations. However, his system requires the user to specify the different input cases for the inverse program Neither Wadler nor Darlington address the issue of the generality of their mechanisms.

Given a program P, we define the inverse of P as P'^1 such that $P^1(P(x)) = x$ for all x in the domain of P. Our problem is to automatically derive P^{1} from P. In other words, starting with an equation of the form $y = F(x)$, we want an equation in the form $F^1(y) = x$ The contribution of this paper is a new method for attacking this problem. It involves successively stripping off the outermost function on the right-hand side and adding the corresponding inverse function to the left-hand side. Each primitive construct in the language requires a separate rule and an additional rule is required to handle recursive and auxiliary function calls. Constructs without a unique inverse are inverted by the introduction of new variables The values of these variables are determined by solving the simultaneous equations generated by this process The language chosen is a minimal subset of pure LISP, including car, cdr, cons, cond, and equal.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA order No 3597, monitored by the Air Force Avionics Laboratory under contract F33615 78 C 1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government

2. The transformation rules

This section describes the transformation rules required to invert programs written in the above LISP subset. For clarity of exposition, the syntax is based on the meta language of McCarthy, et. al. [4]. For example, if

$$z = \mathbf{cons}(x,y),$$

then taking the **car** of both sides we get

$$\mathbf{car}(z) = x.$$

Similarly, if we take the **cdr** of both sides, the result is

$$\mathbf{cdr}(z) = y.$$

Thus, we can replace the first equation by these two derived equations, without loss of information. This rule is quite simple because **cons** is an invertible function.

However, **car** and **cdr** are not invertible functions since we cannot uniquely determine a list given only its **car** or its **cdr**. These functions are inverted by the introduction of new variables. If $z = \mathbf{car}(x)$, then x must be of the form $\mathbf{cons}(z,a)$, where a is the as yet undetermined **cdr** of x. Thus,

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

is replaced by

$$\mathbf{cons}(z,a) = x.$$

Similarly, the equation

$$z = \mathbf{cdr}(x)$$

can be rewritten as

$$\mathbf{cons}(a,z) = x$$

where a is the undetermined **car** of x. The values of the variables are later determined by solving simultaneous equations (see example below).

We also require a rule for inverting the conditional function. A conditional equation of the form

$$y = F(x) = (\text{cond}\ (P(x)\ G(x))\ (Q(x)\ H(x)))$$

is written as

$$y = F(x) = [P(x) \rightarrow G(x);\ Q(x) \rightarrow H(x)]$$

and is transformed into

$$x = F^{-1}(y) = [P(G^{-1}(y)) \rightarrow G^{-1}(y); Q(H^{-1}(y)) \rightarrow H^{-1}(y)]$$

This rule allows us to invert conditional clauses independently. However, if $P(G^{-1}(y))$ and $\sim P(H^{-1}(y))$ and $Q(H^{-1}(y))$ are all true, then the function is not invertible in general. The generalization of the conditional rule to more than two clauses is straightforward.

Finally, if the outermost function on the right-hand side of the current equation is a recursive call to the function we are trying to invert, then we generate a recursive call to the inverse function on the left hand side. In other words, if F(x) is the original function, and the current equation is of the form

$$H(y) = F(G(x))$$

we can replace it with

$$F^{-1}(H(y)) = G(x).$$

where F"^{1} is the name of the inverse function.

This same rule applies when the outermost function on the right-hand side is a user defined auxiliary function. It is replaced by its own inverse on the left hand side and the equation defining the function must then be inverted. Note that if a function has two arguments, its inverse consists of two separate functions, one to produce the first argument and one to produce the second argument. These are distinguished by subscripts in the example below.

3. Example: reverse of a list

As an example of the application of this technique, we invert a program which computes the reverse of a list. Reverse (abbreviated R) takes a list and reverses the order of its top level elements. It uses an auxiliary function, tailcons (abbreviated C), which appends an element to the end of a list. Note that our goal is to express the argument z as a function of R(z) or s. The derivation is shown in figure 1.

The resulting program removes the last element of the list and adds it to the beginning of the reverse of the rest of the list. $C_x^{-1}$is one of the inverses of tailcons. and returns the front of the list excluding the last element. Similarly, C_y^{-1} is the other inverse, and returns the last element of the list. What remains is to derive these two inverses from C(x,y). This is done in figure 2.

4. Observations and current research

An obvious question at this point is how do we know that all the indeterminate variables can be solved for. If a function has a unique inverse, or in other words is one to-one, then the equations that result from applying our rules will have a unique solution. However, our ability to solve those equations will ultimately limit the power of this method.

If a function is not one-to-one, then this technique produces a *partial inverse* in the sense of inverting what can be inverted and indicating exactly what information is missing. For example, consider inverting a program which swaps the elements of a two-element list:

$$SWAP(x) = y = cons(car(cdr(x)),cons(car(x),nil))$$

after inversion becomes

$$SWAP^{-1}(y) = x = cons(car(cdr(y)),cons(car(y),a))$$

This is a partial inverse because the variable a remains indeterminate. The reason is that we neglected to specify that the list contains exactly two elements (i.e. $cdr(cdr(x))=nil$).

1.	$R(z)=s=[z=nil \rightarrow nil;\ T \rightarrow C(R(cdr(z)),car(z))]$	*given*
2.	$z=nil \rightarrow s=nil$	*first clause of cond*
3.	$s=nil \rightarrow z=nil$	*inversion of constant*
4.	$T \rightarrow s = C(R(cdr(z)),car(z))$	*second clause of 1*
5.	$T \rightarrow C_x^{-1}(s) = R(cdr(z))$	*function rule*
6.	$T \rightarrow R^{-1}(C_x^{-1}(s)) = cdr(z)$	*function rule*
7.	$T \rightarrow cons(a,R^{-1}(C_x^{-1}(s))) = z$	*cdr rule*
8.	$T \rightarrow C_y^{-1}(s) = car(z)$	*function rule from 4*
9.	$T \rightarrow cons(C_y^{-1}(s),b) = z$	*car rule*
10.	$T \rightarrow cons(C_y^{-1}(s),R^{-1}(C_x^{-1}(s))) = z$	*solving 7 and 9*
11.	$R^{-1}(s)=z=[s=nil \rightarrow nil;\ T \rightarrow cons(C_y^{-1}(s),R^{-1}(C_x^{-1}(s)))]$	*combining 3 and 10*

Figure 1: Derivation of the Inverse of Reverse

Another observation is that the program for the inverse of reverse is not the same as the original program. Even though the inverse of the reverse *function* is the same function, the inverse of the reverse *program* is in fact the program derived in the example. Inverting the inverse of a program should yield the original program, and it does in the case of reverse.

Current research on this problem is focused on extending this technique to more complex inversion problems In addition, I hope to prove the correctness of the method for as wide a class of programs as possible. In particular, since the inversion algorithm implements a one-to one function, it should be possible for the algorithm to invert itself.

Acknowledgments

I would like to thank Allen Newell, Bill Scherlis, and Herbert Simon for several helpful discussions concerning this research. In addition, Merrick Furst, Michael Horowitz, Greg Korf, Bruce Lucas, and Mark Stehlik read and provided constructive comments on drafts of this paper.

References

[1] Darlington, John.
Program transformation and synthesis, present capabilities.
Technical Report DAI-48, Department of Artificial Intelligence, University of Edinburgh, Sept., 1977.

[2] Dijkstra, E. W.
Program inversion.
Unpublished notes, EWD671, pp. 0 3.

(3] McCarthy, John.
The inversion of functions defined by Turing Machines.
In Shannon, C.E., and J. McCarthy (editors), *Automata Studies,* pages 177-181. Princeton University Press, Princeton, N.J., 1956.

[4] McCarthy, John, et al.
LISP 75 *Programmers Manual.*
MIT. Press,Cambridge, Mass., 1965.

[5] Scherlis, W. L.
Expression procedures and program derivation.
PhD thesis, Department of Computer Science, Stanford University, August, 1980.

[6] Sickel, Sharon.
Invertibility of logic programs.
In *Proceedings of the Fourth Workshop on Automated Deduction,* pages 103-109. Austin, Texas, Feb., 1979.

[7] Wadler, Phil.
What's What?
1980.
internal memorandum, Bell Telephone Laboratories, Murray Hill, N.J.

```
1.  C(x,y)=w=[x=nil→cons(y,nil); T→cons(car(x),C(cdr(x),y))]   given
2.    x=nil→ w = cons(y,nil)                                    first clause of cond
3.      x=nil→ cdr(w) = nil                                     cons rule
4.        cdr(w)=nil→ x = nil                                   inversion of constant

5.      x=nil→ car(w) = y                                       car rule from 2
6.        cdr(w)=nil→ y = car(w)                                see note below

7.    T→ w = cons(car(x),C(cdr(x),y))                           second clause of 1
8.      T→ car(w) = car(x)                                      cons rule
9.        T→ cons(car(w),a) = x                                 car rule

10.     T→ cdr(w) = C(cdr(x),y)                                 cons rule from 7
11.       T→ C_x^{-1}(cdr(w)) = cdr(x)                          function rule
12.         T→ cons(b,C_x^{-1}(cdr(w))) = x                     cdr rule

13.       T→ C_y^{-1}(cdr(w)) = y                               function rule from 10

14.     T→ cons(car(w),C_x^{-1}(cdr(w))) = x                    solving 9 and 12

15. C_x^{-1}(w)=x=[cdr(w)=nil→nil; T→cons(car(w),C_x^{-1}(cdr(w)))]   combining 4 and 14
16. C_y^{-1}(w)=y=[cdr(w)=nil→car(w); T→C_y^{-1}(cdr(w))]              combining 6 and 13
```

Note: the test in equation *6* is the same as that in equation *4* because the original test from equation *2* is a function only of x.

Figure 2: Derivation of the Inverses of Tailcons