Proving Theorems About LISP Functions

ROBERT S. BOYER AND J STROTHER MOORE
University of Edinburgh, Edinburgh, Scotland[*]

Abstract

Program verification is the idea that properties of programs can be precisely stated and proved in the mathematical sense. In this paper, some simple heuristics combining evaluation and mathematical induction are described, which the authors have implemented in a program that automatically proves a wide variety of theorems about recursive LISP functions. The method the program uses to generate induction formulas is described at length. The theorems proved by the program include that `REVERSE` is its own inverse and that a particular `SORT` program is correct. A list of theorems proved by the program is given.

KEY WORDS AND PHRASES: LISP, automatic theorem-proving, structural induction, program verification
CR CATEGORIES: 3.64, 4.22, 5.21

1 Introduction

We are concerned with proving theorems in a first-order theory of lists, akin to the elementary theory of numbers. We use a subset of LISP as our language because recursive list processing functions are easy to write in LISP and because theorems can be naturally stated in LISP; furthermore, LISP has a simple syntax

This work was done on a grant to Professor Bernard Meltzer from the British Science Research Council.

Authors' present addresses: R. S. Boyer, Computer Science Group, Stanford Research Institute, Menlo Park, CA 94025; J Strother Moore, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA 94304.

Journal of the Association for Computing Machinery, Vol. 72, No. 1, January 1975, pp. 129-144.

and is universal in Artificial Intelligence. We employ a LISP interpreter to "run" our theorems and a heuristic which produces induction formulas from information about how the interpreter fails. We combine with the induction heuristic a set of simple rewrite rules of LISP and a heuristic for generalizing the theorem being proved. Our program accepts as input a LISP expression, e.g.

```
(EQUAL (REVERSE (REVERSE A)) A),
```

possibly involving Skolem constants (e.g. A, B, and C throughout this paper) which stand for universally quantified variables ranging over all lists. The program attempts to show that the value of the input expression is always equal to T (whenever the Skolem constants are replaced by arbitrary lists). Theorems we have proved automatically include

```
(EQUAL (REVERSE (REVERSE A)) A)

(IMPLIES (OR (MEMBER A B) (MEMBER A C))
         (MEMBER A (UNION B C)))
```

and

```
(ORDERED (SORT A)),
```

where EQUAL is a LISP primitive function (and built into the theorem-prover) but REVERSE, IMPLIES, OR, MEMBER, UNION, ORDERED, and SORT are defined in LISP by the user of the program. The program uses only its knowledge of the LISP primitives and the LISP definitions supplied by the user. No further information is required of the user. This paper describes many aspects of the program in brevity. A thorough presentation can be found in [22].

2 Our LISP Subset

We use a subset of pure LISP [18] which has as primitives NIL, CONS, CAR, CDR, COND, and EQUAL. With these primitive we can define recursive functions such as those in Appendix A, and we can prove theorems such as those in Appendix B about these functions. Our current theorem-prover assumes that the functions with which it is concerned are total recursive functions, i.e. that they terminate and return an output for any input. (Hence our theorem-prover merely proves "partial correctness.") We do not prove theorems about functions that involve side effects. Our only atom is NIL. We use lists of NILs to represent natural numbers: 0 is NIL, 1 is (CONS NIL NIL), and ADD1 is defined as

```
(LAMBDA (X) (CONS NIL X)).
```

Our arithmetic is thus a version of Peano arithmetic.

NIL is the false truth value. The true truth value in our language is (CONS NIL NIL), which we abbreviate as T.

(CAR (CONS A B)) is defined to be A, and (CDR (CONS A B)) is B. (For the sake of completeness, we define CAR and CDR of NIL to be NIL, but our theorem-prover never uses this definition.)

Our equality primitive is EQUAL rather than EQ. Our conditional primitive, COND, takes three arguments. (COND A B C) in our system is (COND (A B) (T C)) in traditional LISP and means if A is not NIL then B else C.

The user of the theorem-prover supplies function definitions almost exactly as in LISP; for example,

```
(APPEND (LAMBDA (X Y)
        (COND X
           (CONS (CAR X)
              (APPEND (CDR X)
           Y))))).
```

APPEND concatenates its two list arguments.

Our language is simple but powerful enough to encompass primitive recursive number theory. Furthermore, our language could, with suitable conventions, use lists to represent non-NIL atoms, arrays, and strings just as programming languages use bit strings to represent these data types.

3 EVAL

Our LISP interpreter, *EVAL*, is similar in many ways to a normal LISP interpreter; *EVAL* applies function definitions and handles primitives like COND. *EVAL* is recursive; it evaluates arguments before applying and evaluating function definitions. Our *EVAL* has special provisions for handling Skolem constants and terms in which they appear. The following examples illustrate the behavior of our *EVAL*:

```
EVAL( NIL  ) = NIL
EVAL( A  ) = A
EVAL( (CONS A B)  ) = (CONS A B)
EVAL( (CAR (CONS A B))  ) = A
EVAL( (CDR (CAR (CONS A B)))) = (CDR A)
EVAL( (COND (CONS A B) C D)  ) = C
EVAL( (APPEND (CONS A B) C)  ) = (CONS A (APPEND) B C)).
```

The last example is justified because regardless of the values of A, B, and C, the first argument to APPEND is not NIL, so that the COND in the definition of APPEND can be evaluated.

EVAL tries to evaluate (APPEND B C) further but fails because it "recurses into" the Skolem constant B. (See the definition of APPEND above.)

When evaluating the form (FOO $t_1 ... t_n$) where FOO is a defined function, we recursively evaluate the arguments first. Call the value of t_i, t'_i. *EVAL* binds the formal variables of FOO to their values and then evaluates the definition of FOO. If a recursive call of FOO is encountered in this function body the arguments are evaluated as usual. Then, if one of the evaluated arguments is a CAR or CDR expression, it is added to a list called the *BOMBLIST*. In this case the definition of FOO is not reapplied for this recursive call. The current evaluation of the function body is continued in the hopes of adding more terms to the *BOMBLIST*. Finally, *EVAL* returns (FOO $t'_1 ... t'_n$).

Thus, in evaluating (APPEND B C) the recursive call (APPEND (CDR B) C) is encountered in the definition of APPEND. (CDR B) is added to the BOMBLIST indicating recursion on B. Finally, (APPEND B C) is returned as the value of (APPEND B C).

4 Evaluation and Induction

Partial evaluation is sufficient to prove a few trivial theorems; for example,

```
    (EQUAL (APPEND NIL B) B)
```

*EVAL*s to T since partial evaluation of the APPEND yields B, even though the structure of B is not known. However, induction is usually needed to prove even simple theorems about recursive LISP functions.

It is intuitively clear that evaluation and induction are complements. The paradigm for evaluating a simple recursive function FOO is: Evaluate (FOO (CONS A B)) in terms of (FOO B) and handle the NIL case separately. The paradigm for a simple inductive proof that (FOO X) is T for any argument X is: Show that (FOO NIL) is T, and then assuming that (FOO B) is T, show that (FOO (CONS A B)) is T.

In particular, recursion starts with some structure and decomposes it while induction starts with NIL and builds up. This duality can be used to great advantage: Evaluation can be used to reduce the induction conclusion (FOO (CONS A B)) to a statement involving the induction hypothesis (FOO B) provided that the (CONS A B) is one of the structures that FOO decomposes in its recursion.

Suppose that we wish to prove by induction that (FOO X) is T for all X. To show that (FOO NIL) is T, the obvious thing to do is call *EVAL* and let evaluation solve the problem (for example, *EVAL(*(APPEND NIL B) *)* is B). We then assume that (FOO B) is T, and try to show that (FOO (CONS A B)) is T, for a new Skolem constant A. The obvious thing to do now is to call *EVAL* again and let the recursion in FOO decompose the (CONS A B). The result will (hopefully) be some simple expression E, involving (FOO B); we then use the hypothesis that (FOO B) is T to show that E is T. This process is illustrated by the examples in the next two sections.

Of course, if FOO has more than one argument, one must choose which one(s) to induct upon. But the link between evaluation and induction makes the choice obvious: Induct on the structures that are being recursively decomposed by FOO. By choosing those structures we insure that when *EVAL* is called on the induction conclusion, (FOO (CONS A B)), FOO will be able to recurse at least one step and the problem will be reduced by *EVAL* to one involving the induction hypothesis, (FOO B).

However, the terms that FOO is trying to recurse on are just those that generate the "errors" noted earlier. To determine what to induct upon we first *EVAL* the expression (expecting to fail) and then induct upon some term on the *BOMBLIST*, that is, some term which *EVAL* failed to evaluate.

5 A Simple Example of Evaluation and Induction

Suppose we wish to show that

```
(1) (EQUAL (APPEND A (APPEND B C))
           (APPEND (APPEND A B) C))
```

always evaluates to T, for any A, B, and C.

The obvious way to proceed is to *EVAL* the expression and see if it is T. *EVAL* is unable to make any headway in evaluating (1) and simply returns (1) as its answer. However, in attempting to evaluate (1), *EVAL* placed four terms on the *BOMBLIST*. Recall that in the definition of APPEND the first argument is CDRed in the recursion but the second argument is not changed. In formula (1) there are four calls to APPEND. Two recurse upon A, one upon B, and one upon (APPEND A B).

Resorting to induction, we choose to induct upon A (we might have chosen B, but we choose A by "popularity"). First we try the "NIL case" (i.e. (1) with A replaced by NIL):

```
(2) (EQUAL (APPEND NIL (APPEND B C))
           (APPEND (APPEND NIL B) C)).
```

When we *EVAL* this, partial evaluation of the APPENDs makes (2) equivalent to

```
(3) (EQUAL (APPEND B C) (APPEND B C)),
```

since APPEND returns its second argument if the first is NIL. However, (3) is just a partial result, and now that the arguments have been *EVAL*ed, the EQUAL is evaluated to T, since in our LISP two identical expressions return EQUAL results. So the "NIL case" has been shown to be T by evaluation.

Next we must show that

```
(4) (EQUAL (APPEND (CONS A1 A) (APPEND B C))
           (APPEND (APPEND (CONS A1 A) B) C))
```

is always T, if we assume that

```
        (5) (EQUAL (APPEND A (APPEND B C))
                   (APPEND (APPEND A B) C))
```

is T.

But *EVAL* transforms (4) into

```
        (6) (EQUAL (CONS A1 (APPEND A (APPEND B C)))
                   (CONS A1 (APPEND (APPEND A B) C)))
```

and then (from its knowledge of EQUAL) transforms (6) into

```
        (7) (EQUAL (APPEND A (APPEND B C))
                   (APPEND (APPEND A B) C)).
```

But (7) is exactly the same as (5) which we are assuming (inductively) is always T. Hence, by evaluation and the induction hypothesis we have shown that (4), the induction conclusion, is always T. So the associativity of APPEND has been proved. Observe that *EVAL* was responsible for converting the induction conclusion (4) into an expression involving the induction hypothesis (5).

Our program produces precisely this proof. Its only knowledge about APPEND is its LISP definition.

6 Using the Induction Hypothesis and Generalization

Using the induction hypothesis is not always as easy as it was above. A good example occurs in our program's proof of

```
        (8) (EQUAL (REVERSE (REVERSE A)) A),
```

where the definition of REVERSE is

```
        (LAMBDA (X)
                (COND X
                      (APPEND (REVERSE (CDR X))
                              (CONS (CAR X) NIL))
                      NIL)).
```

If we induct on A in (8) we find that the NIL case evaluates to T. We therefore assume (8) as our induction hypothesis and try to prove

```
        (9) (EQUAL (REVERSE (REVERSE (CONS A1 A))) (CONS A1 A)).
```

This evaluates to

```
        (10) (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL)))
                    (CONS A1 A)).
```

We now wish to use the induction hypothesis, (8). Since it is an equality our heuristic is to "cross-fertilize" (10) with it, by replacing the A in the right-hand side of (10) by the left-hand side of (8), giving

```
(11) (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL)))
            (CONS A1 (REVERSE (REVERSE A)))).
```

We then consider (8) to be "used" and throw it away. We must now prove (11).

At this point we note that (REVERSE A) is a subformula which appears on both sides of an EQUAL. Furthermore, from the definition of REVERSE, the program can determine that the output of (REVERSE A) can be any list at all. On these grounds we choose to generalize the theorem to be proved by replacing (REVERSE A) in (11) by a Skolem constant, B, and set out to prove

```
(12) (EQUAL (REVERSE (APPEND B (CONS A1 NIL)))
            (CONS A1 (REVERSE B))).
```

But (12) is easy to prove. *EVAL* tells us to induct on B. The NIL case *EVAL*s to T. Assuming (12) as the induction hypothesis, we *EVAL* the "CONS case":

```
(13) (EQUAL (REVERSE (APPEND (CONS B1 B) (CONS A1 NIL)))
            (CONS A1 (REVERSE (CONS B1 B))))
```

and get

```
(14) (EQUAL (APPEND (REVERSE (APPEND B (CONS A1 NIL)))
                    (CONS B1 NIL))
            (CONS A1 (APPEND (REVERSE B) (CONS B1 NIL)))).
```

We now use our hypothesis, (12), by cross-fertilizing (14) with it, replacing (REVERSE (APPEND B (CONS A1 NIL))) in the left-hand side of (14) by the right-hand side of (12), yielding

```
(15) (EQUAL (APPEND (CONS A1 (REVERSE B)) (CONS B1 NIL))
            (CONS A1 (APPEND (REVERSE B) (CONS B1 NIL)))).
```

Finally, (15) *EVAL*s to T because the left-hand side APPEND evaluates to

```
(CONS A1 (APPEND (REVERSE B) (CONS B1 NIL))),
```

which is the right-hand side, so the EQUAL returns T. The theorem is therefore proved.

Our theorem-prover takes 8 seconds to produce this proof. If the reader thinks that this theorem is utterly trivial, he is invited to try to prove the similar theorem

```
(EQUAL (REVERSE (APPEND A B))
       (APPEND (REVERSE B) (REVERSE A))),
```

which is also proved by the program.

7 A Description of the Program

Besides *EVAL* there are five basic subroutines in our system: *NORMALIZE*, *REDUCE*, *FERTILIZE*, *GENERALIZE*, and *INDUCT*. Below are brief descriptions of these routines.

NORMALIZE applies about ten rewrite rules to LISP expressions. For example,

```
(COND (COND A B C) D E)  becomes
(COND A (COND B D E) (COND C D E)),
```

and `(COND A A NIL)` becomes `A`. Appendix C lists the rewrite rules.

REDUCE attempts to propagate the results of the tests in `COND` statements down the branches of the `COND` tree. Thus,

```
(COND A (COND A B C) (P A))   becomes   (COND A B (P NIL)).
```

FERTILIZE is responsible for “using” the hypothesis of an implication when it is an equality. A theorem of the form $x = y \rightarrow p(y)$ is rewritten to $p(x) \vee x \neq y$. We make fertilizations of the form

$$x = y \rightarrow f(z) = g(y) \text{ becomes } f(z) = g(x) \vee x \neq y.$$

before any other kind. We call such substitutions “cross-fertilizations;” we prefer cross-fertilizations because they frequently allow the proofs we want. After fertilizing we never again look at the equality hypothesis, although we retain it for soundness.

GENERALIZE is responsible for generalizing the theorem to be proved. This is done by replacing some common subformulas in the theorem by new Skolem constants. To prove something of the form $p(f(A)) = q(f(A))$, we try proving $p(B) = q(B)$; and to prove $p(f(A)) \rightarrow q(f(A))$, we try $p(B) \rightarrow q(B)$, where B is a new Skolem constant. However, if the subformula $f(A)$ is of a highly constrained type, for instance, it is always a number, an additional condition is imposed on the new Skolem constant.

If the theorem to be generalized is

```
(EQUAL (ADD (LENGTH A) B) (ADD B (LENGTH A))),
```

GENERALIZE produces as output

```
(COND (LENGTYPE C) (EQUAL (ADD C B) (ADD B C)) T),
```

where `LENGTYPE` is a LISP function written by *GENERALIZE* from the LISP definition of `LENGTH`. In this particular case, the function written by *GENERALIZE* has precisely the definition of `NUMBERP`, namely,

```
(LAMBDA (X)
        (COND X
              (COND (CAR X) NIL (NUMBERP (CDR X)))
              T)).
```

To perform the generalization described in the previous section, *GENERALIZE* wrote the "type function" for `REVERSE`,

```
(LAMBDA (X) T),
```

which was recognized as being no restriction at all and then ignored. The problem of recognizing the output of a recursive function is clearly undecidable and very difficult. To write these type functions, *GENERALIZE* uses some heuristics which are often useful.

INDUCT is the program that embodies our induction heuristic. We now describe the form in which it presents the new induction formula to the other routines and how the induction hypothesis is saved for use.

If the theorem to be proved by induction is `(FOO A)` and *EVAL* indicates that `FOO` recurses on the `CDR` of `A`, the output of *INDUCT* is

```
(COND (FOO NIL)
      (COND (FOO A) (FOO (CONS A1 A)) T)
      NIL),
```

which becomes the theorem to be proved. This is just the LISP expression for

$$\texttt{(FOO NIL)} \land (\texttt{(FOO A)} \to \texttt{(FOO (CONS A1 A))}).$$

The definitions of `AND` and `IMPLIES` are in Appendix A.

The precise form of the induction formula output by *INDUCT* is dictated by the types of "errors" encountered by *EVAL*. For example, if both the `CAR` and the `CDR` of `A` occur on the *BOMBLIST*, then the induction formula is

$$\texttt{(FOO NIL)} \land ((\texttt{(FOO A1)} \land \texttt{(FOO A)}) \to \texttt{(FOO (CONS A1 A))}).$$

For simultaneous recursion on two variables (e.g. `LTE` in Appendix A) and more complicated recursion (e.g. `ORDERED`), *INDUCT* produces appropriate induction formulas. All of this information is collected from the *BOMBLIST* produced by *EVAL*.

8 Control Structure of the Program

The control structure of our system is very simple. To prove that some LISP expression, *THM*, always evaluates to `T`, we execute the following loop:

```
loop:   set  $OLDTHM$  to  $THM$;
        set  $THM$  to  $REDUCE(NORMALIZE(EVAL(THM)))$;
        if  $THM$ = T,  then return;
        if  $THM$  is not equal to  $OLDTHM$,  then goto loop;
        if fertilization applies, then set  $THM$  to  $FERTILIZE(THM)$
        otherwise, if  $THM$  is of the form  (COND $p$ $q$ NIL)
            then set  $THM$  to  (COND  $INDUCT(GENERALIZE(p))$ $q$  NIL)
        otherwise, set  $THM$  to  $INDUCT(GENERALIZE(THM))$;
        goto loop;
```

It should be noted that all of the important control structure is embedded in the LISP expression *THM*. For example, when *INDUCT* needs to prove the conjunction of the NIL case and the induction step, it is actually done by replacing the expression *THM* by a LISP expression which has value T if and only if that conjunction is true. If the NIL case evaluates to T, then *EVAL* returns the second conjunct, which becomes the theorem to be proved.

9 Failures

The program will prove an interesting variety of theorems, as illustrated by Appendix B. However, the system is far from being a practical tool for program verification because it fails to prove many interesting theorems.

We believe that the current program is incapable of proving theorems more difficult — in an intuitive sense — than the correctness of the list sorting function in Appendix A. To prove that the output of SORT is ORDERED requires the generation and proof of two lemmas: that ADDTOLIST produces an ordered list when its second argument is ordered, and that the function LTE (less than or equal) has the property that (for all X and Y), X is LTE Y, or Y is LTE X. The program generates these two lemmas and proves them by induction. (Note: In order to prove the correctness of SORT, one must not only show that its output is ORDERED but that the output is a permutation of the input. The program proves this as well.)

Although the correctness of SORT is interesting, computers are full of programs much more complicated. Thus the statement that the above theorem is essentially at the limit of the current program's power is especially disturbing to those interested in practical applications.

Below we discuss two common causes of the theorem-prover's failure to prove many theorems.

The first cause is simply that the induction mechanism is too simple. As mathematicians have known for years, it is often not at all obvious what one must assume in order to prove a theorem inductively. The current program suffers from making the wrong hypotheses, even when the right ones are obvious.

Consider the function

```
(REVERSE1 (LAMBDA (X Y)(COND X
                             (REVERSE1 (CDR X)
                                       (CONS (CAR X) Y))
                             Y))).
```

Note that as the function recursively destroys X, it recursively constructs Y.

If a theorem involving (REVERSE1 A B) is to be proved by induction on A, the conclusion will involve the term (REVERSE1 (CONS A1 A) B). When this is symbolically evaluated it yields (REVERSE1 A (CONS A1 B)). Therefore, this term should be involved in the hypothesis if the hypothesis is to be useful. However, the current program supplies the term (REVERSE1 A B) in the hypothesis, because it fails to note the way the second argument is being used.

A second cause of failure involves generalization and induction. To prove the theorem

```
    (IMPLIES (SUBSET A A) (SUBSET A (CONS A1 A)))
```

requires the generalization

```
    (IMPLIES (SUBSET A B) (SUBSET A (CONS A1 B))).
```

The program can easily prove the generalization, but it fails to prove the original theorem. Recognizing the need for the generalization and deciding which variables to "separate" is difficult.

We feel that despite the shortcomings of the current program, the approach to program verification discussed here is promising. Just as it is difficult if not impossible to integrate symbolically certain formulas without certain 'tricks," it is difficult if not impossible to prove certain theorems without certain "tricks." The program is an implementation of some of these tricks and we believe it is possible to discover more and implement them in a way that is within the spirit of the current approach, if not within the framework of the current program.

10 Conclusion

We find it natural to use the routines *EVAL*, *NORMALIZE*, and *REDUCE* both to rewrite LISP expressions and prove theorems. Our experience confirms, and was motivated by, a conviction that proofs and computations are essentially similar. This conviction was inspired by conversations with Bob Kowalski and Pat Hayes, and the beauty of LISP. Our program is in the style of theorem-proving programs written by Bledsoe [1, 2].

We would like to note that our program uses no search and has no knowledge of user functions other than their definitions. Consequently our theorem-prover frequently reproves simple facts like the associativity of APPEND.

11 Related Work

An excellent survey of the various methods for verifying programs is presented in Manna, Ness, and Vuillemin, 1972 [16].

Brotz, 1973 [3] has implemented an arithmetic theorem-prover very similar to ours. His system generates its own induction formulas and uses the generalization heuristic we use (without "type functions"). He inducts upon the rightmost Skolem constant appearing in the statement of the theorem rather than using *EVAL* and the *BOMBLIST* as we do. His heuristic will always choose a term recursed upon (due to restrictions on the forms of recursive equations allowed), but it will not always choose the one we choose.

Our program uses structural induction, which was introduced into the literature by Burstall, 1969 [4], although it was used earlier by McCarthy and Painter, 1967 [19] in a compiler correctness proof. Common alternative inductive methods for recursive languages are computational induction (Park, 1969 [24]) and recursion induction (McCarthy, 1963 [17]). Both are essentially induction on the depth of function calls. Milner, 1972 [20] and Milner and Weyhrauch, 1972 [21] describe a proof checker for Scott's Logic for Computable Functions (Scott, 1970 [25]) which uses computational induction. The most commonly used method is for flow diagram languages and was suggested by Naur, 1966 [23] and Floyd, 1967 [10]. In this approach, inductive assertions are attached to points in a program and are used to generate "verification conditions," which are theorems that must be proved to establish the correctness of the program. King 1969 [15], Good, 1970 [12], Cooper, 1971 [5], Gerhart, 1972 [11], Deutsch, 1973 [7], Igarashi, London, and Luckham, 1973 [13], Elspas, Levitt, and Waldinger, 1973 [9], and Topor and Burstall, 1973 [26] have implemented systems which use this method for languages which include assignments (possibly to arrays), jumps or loops, and defined procedure calls. These programs require the user to invent inductive assertions. Elspas, 1972 [8], Wegbreit, 1973 [27], and Katz and Manna, 1973 [14] present heuristics for generating inductive assertions automatically. Darlington and Burstall, 1973 [6] describe a system which will take functions such as the ones in our LISP subset and write equivalent programs which are more efficient. This system will replace recursion by iteration, merge loops, and use data structures (destructively) when permitted.

Appendix A. Function Definitions

Appendix A contains the definition of the LISP functions we use in the proofs of the theorems in Appendix B. The program automatically proves all of the theorems in Appendix B. The average time to prove each theorem is 8 seeonds on an ICL 4130 using POP-2. The time is almost completely spent in POP-2 list processing, where the time for a `CONS` is 400 microseconds, and for a `CAR` and `CDR` is 50 microseconds.

```
(ADD (LAMBDA (X Y)
             (COND (ZEROP X)
                   (LENGTH Y)
                   (ADD1 (ADD (SUB1 X)
                              Y)))))
(ADD1 (LAMBDA (X)
              (CONS NIL X)))
(ADDTOLIST (LAMBDA (X Y)
                   (COND Y (COND (LTE X (CAR Y))
                                 (CONS X Y)
                                 (CONS (CAR Y)
                                       (ADDTOLIST X (CDR Y))))

                         (CONS X NIL))))
(AND (LAMBDA (X Y)
             (COND X (COND Y T NIL)
                   NIL)))
(APPEND (LAMBDA (X Y)
                (COND X (CONS (CAR X)
                              (APPEND (CDR X)
                                      Y))
                      Y)))
(ASSOC (LAMBDA (X Y)
               (COND Y (COND (CAR Y)
                             (COND (EQUAL X (CAR (CAR Y)))
                                   (CAR Y)
                                   (ASSOC X (CDR Y)))
                             (ASSOC X (CDR Y)))
                     NIL)))
(BOOLEAN (LAMBDA (X)
                 (COND X (EQUAL X T)
                       T)))
(CDRN (LAMBDA (X Y)
              (COND Y (COND (ZEROP X)
                            Y
                            (CDRN (SUB1 X)
                                  (CDR Y)))
                    NIL)))
(CONSNODE (LAMBDA (X Y)
                  (CONS NIL (CONS X Y))))
(COPY (LAMBDA (X)
              (COND X (CONS (COPY (CAR X))
                            (COPY (CDR X)))
                    NIL)))
(COUNT (LAMBDA(X Y)
              (COND Y (COND (EQUAL X (CAR Y))
                            (ADD1 (COUNT X (CDR Y)))
                            (COUNT X (CDR Y)))
                    0)))
(DOUBLE (LAMBDA (X)
                (COND (ZEROP X)
                      0
                      (ADD 2 (DOUBLE (SUB1 X))))))
(ELEMENT (LAMBDA (X Y)
                 (COND Y (COND (ZEROP X)
                               (CAR Y)
                               (ELEMENT (SUB1 X)
                                        (CDR Y)))
                       NIL)))
(EQUALP (LAMBDA (X Y)
                (COND X (COND Y (COND(EQUALP (CAR X)
                                             (CAR Y))
                                     (EQUALP (CDR X)
                                             (CDR Y))
                                     NIL)
                              NIL)
                      (COND Y NIL T))))
(EVEN1 (LAMBDA (X)
               (COND (ZEROP X)
                     T
                     (ODD (SUB1 X)))))
(EVEN2 (LAMBDA (X)
               (COND (ZEROP X)
                     T
                     (COND (ZEROP (SUB1 X))
                           NIL
                           (EVEN2 (SUB1(SUB1 X)))))))
(FLATTEN (LAMBDA(X)
                (COND (NODE X)
                      (APPEND (FLATTEN (CAR (CDR X)))
                              (FLATTEN (CDR (CDR X))))
                      (CONS X NIL))))
(GT (LAMBDA (X Y)
            (COND (ZEROP X)
                  NIL
                  (COND (ZEROP Y)
                        T
                        (GT (SUB1 X)
                            (SUB1 Y))))))
(HALF (LAMBDA (X)
              (COND (ZEROP X)
                    0
                    (COND (ZEROP (SUB1 X))
                          0
                          (ADD1 (HALF (SUB1 (SUB1 X))))))))
(IMPLIES (LAMBDA (X Y)
                 (COND X (COND Y T NIL)
                       T)))
(INTERSECT (LAMBDA (X Y)
                   (COND X (COND (MEMBER (CAR X)
                                         Y)
                                 (CONS (CAR X)
                                       (INTERSECT (CDR X)
                                                  Y))
                                 (INTERSECT (CDR X)
                                            Y))
                         NIL)))
(LAST (LAMBDA (X)
              (COND X (COND (CDR X)
                            (LAST (CDR X))
                            (CAR X))
                    NIL)))
(LENGTH (LAMBDA (X)
                (COND X (ADD1 (LENGTH (CDR X)))
                      0)))
(LIT (LAMBDA (X Y Z)
             (COND X (APPLY Z (CAR X)
                            (LIT (CDR X)
                                 Y Z))
                   Y)))
(LTE (LAMBDA (X Y)
             (COND (ZEROP X)
                   T
                   (COND (ZEROP Y)
                         NIL
                         (LTE (SUB1 X)
                              (SUB1 Y))))))
(MAPLIST (LAMBDA (X Y)
                 (COND X (CONS (APPLY Y (CAR X))
                               (MAPLIST (CDR X)
                                        Y))
                       NIL)))
(MEMBER (LAMBDA (X Y)
                (COND Y (COND (EQUAL X (CAR Y))
                              T
                              (MEMBER X (CDR Y)))
                      NIL)))
(MONOT1 (LAMBDA (X)
                (COND X (COND (CDR X)
                              (COND (EQUAL (CAR X)
                                           (CAR (CDR X)))
                                    (MONOT1 (CDR X))
                                    NIL)
                              T)
                      T)))
(MONOT2 (LAMBDA (X Y)
                (COND Y (COND (EQUAL X (CAR Y))
                              (MONOT2 X (CDR Y))
                              NIL)
                      T)))
(MONOT2P (LAMBDA (X)
                 (COND X (MONOT2 (CAR X)
                                 (CDR X))
                       T)))
(MULT (LAMBDA (X Y)
              (COND (ZEROP X)
                    0
                    (ADD Y (MULT (SUB1 X)
                                 Y)))))
(NODE (LAMBDA (X)
              (COND X (COND (CAR X)
                            NIL
                            (COND (CDR X)
                                  T NIL))
                    NIL)))
(NOT (LAMBDA (X)
             (COND X NIL T)))
(NUMBERP (LAMBDA (X)
                 (COND X (COND (CAR X)
                               NIL
                               (NUMBERP (CDR X)))
                       T)))
(OCCUR (LAMBDA (X Y)
               (COND (EQUAL X Y)
                     T
                     (COND Y (COND (OCCUR X (CAR Y))
                                   T
                                   (OCCUR X (CDR Y)))
                           NIL))))
(ODD (LAMBDA (X)
             (COND (ZEROP X)
                   NIL
                   (EVEN1 (SUB1 X)))))
(OR (LAMBDA (X Y)
            (COND X T (COND Y T NIL))))
(ORDERED (LAMBDA (X)
                 (COND X (COND (CDR X)
                               (COND (LTE (CAR X)
                                          (CAR (CDR X)))
                                     (ORDERED (CDR X))
                                     NIL)
                               T)
                       T)))
(PAIRLIST (LAMBDA (X Y)
                  (COND X (COND Y (CONS (CONS (CAR X)
                                              (CAR Y))
                                        (PAIRLIST (CDR X)
                                                  (CDR Y)))
                                (CONS (CONS (CAR X)
                                            NIL)
                                      (PAIRLIST (CDR X)
                                                NIL)))
                        NIL)))
(REVERSE (LAMBDA (X)
                 (COND X (APPEND (REVERSE (CDR X))
                                 (CONS (CAR X)
                                       NIL))
                       NIL)))
(SORT (LAMBDA (X)
              (COND X (ADDTOLIST (CAR X)
                                 (SORT (CDR X)))
                    NIL)))
(SUB1 (LAMBDA (X)
              (CDR X)))
(SUBSET (LAMBDA (X Y)
                (COND X (COND (MEMBER (CAR X)
                                      Y)
                              (SUBSET (CDR X)
                                      Y)
                              NIL)
                      T)))
(SUBST (LAMBDA (X Y Z)
               (COND (EQUAL Y Z)

                     X
                     (COND Z (CONS (SUBST X Y (CAR Z))
                                   (SUBST X Y (CDR Z)))
                           NIL))))
(SWAPTREE (LAMBDA (X)
                  (COND (NODE X)
                        (CONSNODE (SWAPTREE (CDR (CDR X)))
                                  (SWAPTREE (CAR (CDR X))))
                        X)))
(TIPCOUNT (LAMBDA (X)
                  (COND (NODE X)
                        (ADD (TIPCOUNT (CAR (CDR X)))
                             (TIPCOUNT (CDR (CDR X))))
                        1)))
(UNION (LAMBDA (X Y)
               (COND X (COND (MEMBER (CAR X)
                                     Y)
                             (UNION (CDR X)
                                    Y)
                             (CONS (CAR X)
                                   (UNION (CDR X)
                                          Y)))
                     Y)))
(ZEROP (LAMBDA (X)
               (EQUAL X 0)))
```

Appendix B. Theorems Proved

```
(EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C))
(IMPLIES (EQUAL (APPEND A B) (APPEND A C)) (EQUAL B C))
(EQUAL (LENGTH (APPEND A B)) (LENGTH (APPEND B A)))
(EQUAL (REVERSE (APPEND A B)) (APPEND (REVERSE B) (REVERSE A)))
(EQUAL (LENGTH (REVERSE D)) (LENGTH D))
(EQUAL (REVERSE (REVERSE A)) A)
(IMPLIES A (EQUAL (LAST (REVERSE A)) (CAR A)))
(IMPLIES (MEMBER A B) (MEMBER A (APPEND B C)))
(IMPLIES (MEMBER A B) (MEMBER A (APPEND C B)))
(IMPLIES (AND (NOT (EQUAL A (CAR B))) (MEMBER A B)) (MEMBER A (CDR B)))
(IMPLIES (OR (MEMBER A B) (MEMBER A C)) (MEMBER A (APPEND B C)))
(IMPLIES (AND (MEMBER A B) (MEMBER A C)) (MEMBER A (INTERSECT B C)))
(IMPLIES (OR (MEMBER A B) (MEMBER A C)) (MEMBER A (UNION B C)))
(IMPLIES (SUBSET A B) (EQUAL (UNION A B) B))
(IMPLIES (SUBSET A B) (EQUAL (INTERSECT A B) A))
(EQUAL (MEMBER A B) (NOT (EQUAL (ASSOC A (PAIRLIST B C)) NIL)))
(EQUAL (MAPLIST (APPEND A B) C) (APPEND (MAPLIST A C) (MAPLIST B C)))
(EQUAL (LENGTH (MAPLIST A B)) (LENGTH A))
(EQUAL (REVERSE (MAPLIST A B)) (MAPLIST (REVERSE A) B))
(EQUAL (LIT (APPEND A B) C D) (LIT A (LIT B C D) D))
(IMPLIES (AND (BOOLEAN A) (BOOLEAN B))
         (EQUAL (AND (IMPLIES A B) (IMPLIES B A)) (EQUAL A B)))
(EQUAL (ELEMENT B A) (ELEMENT (APPEND C B) (APPEND C A)))
(IMPLIES (ELEMENT B A) (MEMBER (ELEMENT B A) A))
(EQUAL (CDRN C (APPEND A B)) (APPEND (CDRN C A) (CDRN (CDRN A C) B)))
(EQUAL (CDRN (APPEND B C) A) (CDRN C (CDRN B A)))
(EQUAL (EQUAL A B) (EQUAL B A))
(IMPLIES (AND (EQUAL A B) (EQUAL B C)) (EQUAL A C))
(IMPLIES (AND (BOOLEAN A) (AND (BOOLEAN B) (BOOLEAN C)))
         (EQUAL (EQUAL (EQUAL A B) C) (EQUAL A (EQUAL B C))))
(EQUAL (ADD A B) (ADD B A))
(EQUAL (ADD A (ADD B C)) (ADD (ADD A B) C))
(EQUAL (MULT A B) (MULT B A))
(EQUAL (MULT A (ADD B C)) (ADD (MULT A B) (MULT A C)))
(EQUAL (MULT A (MULT B C)) (MULT (MULT A B) C))
(EVEN1 (DOUBLE A))
(IMPLIES (NUMBERP A) (EQUAL (HALF (DOUBLE A)) A))
(IMPLIES (AND (NUMBERP A) (EVEN1 A)) (EQUAL (DOUBLE (HALF A)) A))
(EQUAL (DOUBLE A) (MULT 2 A))
(EQUAL (DOUBLE A) (MULT A 2))
(EQUAL (EVEN1 A) (EVEN2 A))
```

```
(GT (LENGTH (CONS A B)) (LENGTH B))
(IMPLIES (AND (GT A B) (GT B C)) (GT A C))
(IMPLIES (GT A B) (NOT (GT B A)))
(LTE A (APPEND B A))
(OR (LTE A B) (LTE B A))
(OR (GT A B) (OR (GT B A) (EQUAL (LENGTH A) (LENGTH B))))
(EQUAL (MONOT2P A) (MONOT1 A))
(ORDERED (SORT A))
(IMPLIES (AND (MONOT1 A) (MEMBER B A)) (EQUAL (CAR A) B))
(LTE (CDRN A B) B)
(EQUAL (MEMBER A (SORT B)) (MEMBER A B))
(EQUAL (LENGTH A) (LENGTH (SORT A)))
(EQUAL (COUNT A B) (COUNT A (SORT B)))
(IMPLIES (ORDERED A) (EQUAL A (SORT A)))
(IMPLIES (ORDERED (APPEND A B)) (ORDERED A))
(IMPLIES (ORDERED (APPEND A B)) (ORDERED B))
(EQUAL (EQUAL (SORT A) A) (ORDERED A))
(LTE (HALF A) A)
(EQUAL (COPY A) A)
(EQUAL (EQUALP A B) (EQUAL A B))
(EQUAL (SUBST A A B) B)
(IMPLIES (MEMBER A B) (OCCUR A B))
(IMPLIES (NOT (OCCUR A B)) (EQUAL (SUBST C A B) B))
(EQUAL (EQUALP A B) (EQUALP B A))
(IMPLIES (AND (EQUALP A B) (EQUALP B C)) (EQUALP A C))
(EQUAL (SWAPTREE (SWAPTREE A)) A)
(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
(EQUAL (LENGTH (FLATTEN A)) (TIPCOUNT A))
```

Appendix C. Rewrite Rules Applied by Normalize

In the rules below, lower-case letters represent arbitrary forms. Forms matching those on the left-hand side of thc arrows are replaced by the appropriate instances of the forms on the right. *IDENT* is a routine which takes two terms as arguments and returns "equal" if they are syntatically identical (such as (CONS A B) and (CONS A B), or (CONS NIL NIL) and 1), "unequal" if they are obviously unequal (such as (CONS A B) and NIL, or (CONS A B) and A), or "unknown". *BOOLEAN* is a routine which returns true or false depending upon whether its argument is Boolean. *BOOLEAN* handles recursive functions by inspecting their definitions with an inductive assumption that any recursive calls are to be considered Boolean. *NORMALIZE* rewrites the arguments to the term it is given before rewriting the top-level expression. Finally, any rule involving EQUAL has a symmetric version not presented in which the arguments to the EQUAL have been interchanged.

(EQUAL x y) ⇒ T, if *IDENT(* x, y*)* = "equal"
(EQUAL x y) ⇒ NIL, if *IDENT(* x, y*)* = "unequal"

```
(EQUAL x T) ⇒ x,  if  BOOLEAN( x)
(EQUAL (EQUAL x y) z) ⇒ (COND (EQUAL x y) (EQUAL z T)(COND z NIL T))
(COND (CONS u v) x y) ⇒ x
(COND NIL x y) ⇒ y
(COND x T NIL) ⇒ x,  if  BOOLEAN( x)
(COND x y y) ⇒ y
(COND x x NIL) ⇒ x
(f x... (COND y u v) ...z) ⇒ (COND y (f x...u...z)(f x...v...z))
```

ACKNOWLEDGMENT. We are grateful to Professor Bernard Meltzer for his support and supervision.

REFERENCES

1. Bledsoe, W. W. Splitting and reduction heursitics in automatic theorem proving. *Artif. Intel. 2*, 1 (1971), 55 77.

2. Bledsoe, W. W., Boyer, R. S., and Henneman, W. H. Computer proofs of limit theorems. *Artif. Intel. 3* (1972), 27-60.

3. Brotz, D. Proving theorems by mathematical induction. Ph.D. Th., Comput. Sci. Dep., Stanford U., Stanford, Calif., 1973.

4. Burstall, R. M. Proving properties of programs by structural induction. *Comput. J. 12* (1969), 41-48.

5. Cooper, D. Programs for mechanical program verification. In *Machine Intelligence 6*, B. Meltzer and D. Michie, Eds., Edinburgh U. Press, Edinburgh, 1971, pp. 43-59.

6. Darlington, J., and Burstall, R. M. A system which automatically improves programs. Proc. Internat. Joint Conf. on Artif. Intell., 1973, pp. 479-485.

7. Deutsch, L. P. An interactive program verifier. Ph.D. Th., Comput. Sci. Dep., U. of California, Berkeley, Calif., 1973.

8. Elspas, B. The semiautomatic generation of inductive assertions for program correctness proofs. Rep. No. 55, Seminar, Des Instituts fur Theorie der Automaten und Schaltnetzwerke, Gesellschaft fur Mathematik und Datenverarbeitung, Bonn, Aug. 21,1972.

9. Elspas, B., Levitt, K. N., and Waldinger, R. J. An interactive system for the verification of computer programs. Final rep., Project 1891, Stanford Res. Inst., Menlo Park, Calif., 1973.

10. Floyd, R. W. Assigning meaning to programs. Proceedings of a Symposium in Applied Mathematics, Vol. 19: Mathematical Aspects of Computer Science, J. T. Schwartz, Ed., Amer. Math. Soc., Providence, R. I., 1967, pp. 19-32.

11. Gerhart, S. Verification of APL programs. Ph.D. Th., Carnegie-Mellon U., Pittsburgh, Pa., 1972.

12. Good, D. Toward a man-machine system for proving program correctness. Ph.D. Th., Dep. of Comput. Sci., U. of Wisconsin, Madison, Wis., 1970.

13. Igarashi, S., London, R. L., and Luckham, D. C. Automatic program verification I: A logical basis and its implementation. Rep. 200, Stanford Artif. Intel. Lab., Stanford, Calif., 1973.

14. Katz, S. M., and Manna, Z. A heuristic approach to program verification. Proc. Int'l Joint Conf. on Artif. Intell., 1973, pp. 500-512.

15. King, J. A program verifier. Ph.D. Th., Carnegie-Mellon U., Pittsburgh, Pa., 1969.

16. Manna, Z., Ness, S., and Vuillemln, J. Inductive methods for proving properties of programs. Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1 (Jan. 1972), pp. 27-50.

17. McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds., North-Holland, Amsterdam, 1963, pp. 33-70.

18. McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* M.I.T. Press, Cambridge, Mass., 1962.

19. Mccarthy, J., and Painter, J. A. Correctness of a compiler for arithmetic expressions. Proceedings of a Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, Schwartz, J. T., Ed., Amer. Math. Soc., Providence, R. I., 1967, pp. 33-41.

20. Milner, R. Implementation and application of Scott's logic for computable functions. Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1 (Jan. 1972), pp. 1-6.

21. Milner, R., and Weyhrauch, R. Proving compiler correctness in a mechanized logic In *Machine Intelligence 7*, B. Meltzer and D. Michie, Eds., Edinburgh U. Press, Edinburgh, 1972, pp. 5170.

22. Moore, J S. Computational logic: Structure sharing and proof of program properties. Ph.D. Th., Dep. of Computational Logic, School of Artif. Intel., U. of Edinburgh, Edinburgh, 1973.

23. Naur, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310-316.

24. Park, D. Fixpoint induction and proofs of program properties. In *Machine Intelligence 5*, B. Meltzer and D. Michie, Eds., Edinburgh U. Press, Edinburgh, 1969, pp. 59-78.

25. Scott, D. Outline of a Mathematical Theory of Computation. Tech. Monograph PRG-2, Programming Res. Group, Oxford U. Computing Lab., Nov. 1970.

26. Topor, R., and Burstall, R. M. Private communication (1973).

27. Wegbreit, B. The synthesis of loop predicates. *Comm. ACM 17*, 2(Feb. 1974),102-112.

received september 1993; revised april 1994