A LEMMA DRIVEN AUTOMATIC THEOREM PROVER FOR RECURSIVE FUNCTION THEORY

Robert S. Boyer

J Strother Moore

Computer Science Laboratory
SRI International
Menlo Park, CA 94025

ABSTRACT

We describe work in progress on an automatic theorem prover for recursive function theory that we intend to apply in the analysis (including verification and transformation) of useful computer programs. The mathematical theory of our theorem prover is extendible by the user and serves as a logical basis of program specification (analogous to, say, the predicate calculus). The theorem prover permits no interaction once given a goal, but many aspects of its behavior are influenced by previously proved results. Thus, its performance on difficult theorems can be radically improved by having it first prove relevant lemmas. We describe several ways that the theorem prover employs such lemmas. Among the interesting theorems proved are the correctness of a simple optimizing compiler for expressions and the correctness of a "big number" addition algorithm.

The research reported here has been supported by the Office of Naval Research under Contract NOOO14-75-C-0816, the National Science Foundation under Grant DCR72-03737A01, and the Air Force Office of Scientific Research under Contract F44620-73-C-0068.

I INTRODUCTION

This paper presents work in progress on an automatic theorem prover designed for use in the analysis of computer programs. Like our earlier LISP theorem prover [1], [2], [3], [4], the new system proves theorems in a version of recursive function theory; however, the new version, which is considerably richer than the old, is axiomatically extendible and can serve as a logical basis of program specification (analogous to predicate calculus). Of course, the new theory is tailored to programming languages in the sense that it is easy to model, say, arrays and other data objects in it. But exactly how one attaches meaning to programs (e.g. with the functional approach of McCarthy and Burstall or with the inductive assertion approach of Floyd and Hoare). is the business of the program verification system, not the mathematical theory or the theorem proving system. (For example, we have recently formalized in our theory a version of the Pernas-Robinson [5] hierarchical program design methodology using the Floyd approach to program correctness.)

The approach we have taken towards automating proofs in the theory is to model our system on the way we prove theorems: we simplify the conjecture whenever possible, applying relevant axioms and theorems; we split the problem into as many independent subgoals as possible; we carefully apply various heuristics, such as equality substitutions, elimination of undesirable or irrelevant terms, and generalization; and then we formulate the cleanest possible induction for each subgoal based on a thorough analysis of how the functions behave. We also monitor our progress by comparing subgoals with one another. For example, we look for indications of looping and for opportunities to subsume a subgoal with a more general subgoal. Our basic approach to theorem proving, then, is much in the style of Bledsoe (as described, for example, in [6] and [7]).

We designed the new system to be entirely automatic. However, unlike our earlier LISP theorem prover, which was built to demonstrate completely automatic induction proof techniques, the new system can take advantage of previously proved results to alter dramatically its behavior. With proper "training" it can construct proofs of quite complicated theorems. The basic idea is to be able to do simple proofs automatically while allowing the user to structure difficult proofs by stating relevant lemmas to be proved beforehand. The optimizing compiler proof, discussed below, illustrates how the user can structure a "difficult" theorem by suggesting several "simple" lemmas.

Aubin [8] and Cartwright [9] have recently produced related automatic theorem provers for versions of recursive function theory. We highly recommend their theses to anyone interested in mechanizing recursive function theory.

This paper is organized as follows. We first present a description of the theory with which the theorem prover deals. We then describe in some detail the system's automatic proof of the correctness of a simple optimizing compiler for expressions. Next we discuss some of the ways lemmas are employed to guide the theorem prover. That section discusses lemma driven simplification, generalization, and an interesting use of lemmas to Justify inductions. The discussion of induction summarizes one of the system's proofs about "big

number" arithmetic. The paper has two appendices. The first presents the definitions of recursive functions referred to in the paper. The second presents the system's output during a proof of a simple theorem about a tree-flattening algorithm. We include this proof both as an example of the theorem prover's behavior end because the recursion (8nd induction) in the tree-flattening algorithm is similar to that used by the compiler. Because of space limitations we cannot ennumerate the theorems the system has proved. Such a list will be sent upon request. We also welcome requests for the source code of our system.

II THE NEW THEORY

The theory we now use admits arbitrary recursive definitions. (Our earlier LISP theorem prover is limited to primitive recursion.) We regard definitional equations as axioms which are used only to expand function calls. (One might use recursive functions as a logical basis for induction arguments. However, we derive our Induction principles solely from proven lemmas in a way described below. Of course, we use the recursive structure of the functions in our theorems to suggest which principle of induction Is most appropriate.)

The domain of objects is partitioned into an infinity of disjoint "type classes." The user is free to define axiomatically the properties of any class. Because the free variables in a conjecture are understood to range over all objects (even objects not yet distinguished by the axiomatization of their containing classes), theorems remain theorems even when a new class is axiomatized. Of course, variables in a theorem may be constrained with axiomatically or recursively defined predicates. Unlike the languages of Aubin [8] and Cartwright [9], our language is not typed; we do not have typed variables in our theorems or type constraints on the input or output of our functions.

The initial objects are TRUE and FALSE, the sole members of their respective classes. The initial functions are IF and EQUAL. (IF x y z) is z if x is FALSE and y otherwise. (EQUAL x y) is TRUE if x is y and FALSE otherwise.

Arithmetic over the non-negative integers, the literal atoms, lists, push-down stacks, characters, and strings of characters *&re* examples of classes that have been added axiomatically. There is a (meta-) facility called ADD.SHELL for automatically axiomatizing new classes of objects with properties virtually identical to Burstall's structures [10] and close to the style of Clark and Tarnlund [11].

The theory also contains a version of the principle of induction called the Generalized Principle of Induction (Noetherian induction) in Burstall [10]. The principle allows one to induct over any well-founded partial ordering.

III AN EXAMPLE

Let us now consider the system's proof of the correctness of a very simple optimizing compiler for expressions. Although this proof fails to exercise many of the new heuristics in the system, it does illustrate the use of lemmas and the system's general competence at handling significantly larger problems than our earlier LISP theorem prover could. A detailed presentation of the proof (complete with the set of all axioms used, the system's very readable commentary on the proof, end a discussion of several bugs discovered in earlier attempts to write the compiler) is available in [12].

Suppose we have axiomatically introduced as distinct classes the non-negative integers, atomic symbols, list structures, end push-down stacks. For simplicity, suppose we want to optimize and compile numerically valued expressions composed of integers, variables, and binary function symbols. We will represent such expressions as S-expressions (e.g. "(3 ♦ x) • y" is represented as (TIMES (PLUS 3 x) y)). The recursive predicate FORMP recognizes such forms. (All of the functions mentioned in this section are formally defined in Appendix A.)

The main theorem we wish to prove is that executing the (optimized end) compiled code for a form in the context of some stack and environment is equivalent just to pushing onto the stack the value of the unoptimized form in the environment. That is:

```
CORRECTNESS.OF.OPTIMIZING.COMPILER:
(FORMP x) -> (EXEC (COMPILE x) pds envrn)

                (PUSH (EVAL x envrn) pds).
```

The compiler, COMPILE, works in two passes. It first uses OPTIMIZE to perform constant folding (e.g., (TIMES (PLUS 3 4) y) is optimized to (TIMES 7 y) if (APPLY 'PLUS 3 4 were to have the value 7). After optimizing, the compiler calls CODEGEN to generate a list of instructions to be executed (in REVERSE order). (CODEGEN form insj generates a list of instructions for form, treating ins as the list of instructions previously "laid down". If form is a number or variable, CODEGEN lays down a PUSHI or PUSHV instruction (by consing it onto ins). (PUSHI n) causes the "hardware" to push n onto its stack. (PUSHV x) causes it to push the value of the variable x in the current environment. If the form is not a number or variable, then CODEGEN assumes it is an S-expression such as (fn argl arg2). It first lays down the code for argl, then the code for arg2, and then the instruction fn, which means to the hardware "pop two things, apply fn to them, and push the result."

The reader is encouraged to inspect the definition of CODEGEN in Appendix A. Note, in particular, that the compilation of arg2 takes place with ins being the sequence produced by a recursive call of CODEGEN on argl.

As an example of CODEGEN, its (reversed) output for the form (TIMES 3 (PLUS x y)) is:

```
((PUSH1 3)
 (PUSHV x)
 (PUSHV y)
 PLUS
 TIMES).
```

The semantics of expressions (in an environment, envrn, specifying the values of variables) is defined by EVAL. The value of a number is itself. The value of a variable, x, is (GETVALUE x envrn), where GETVALUE is an unspecified function. The value of (fn arg1 arg2) is the result of APPLYing fn to the values of arg1 and arg2, where APPLY is a numerically valued function that is otherwise unspecified.

Finally, EXEC is a formalization of a machine's instruction fetch and execute cycle. EXEC takes three arguments: an instruction sequence, a push-down stack, and an environment. Like EVAL, EXEC uses GETVALUE and APPLY. (Regardless of the semantics of GETVALUE and APPLY, the optimizing compiler must cause EXEC to compute the same value for forms as EVAL would.) EXEC returns the push-down stack after the last instruction has been executed.

In order to prove the main theorem, we (the users of the theorem prover) structure the proof as follows: First, show that OPTIMIZE does not change the value (a la EVAL) of the form optimized. Independently show that when CODEGEN is given any form whatsoever it generates code that, when executed, pushes the value of the form on the stack. To use these two lemmas to prove the main theorem we must only then establish that OPTIMIZE does indeed give CODEGEN a form to work on. Formally, the three "simple" lemmas are:

```
CORRECTNESS.OF.OPTIMIZE:
(FORMP x) -> (EVAL (OPTIMIZE x) envrn)
                        =
             (EVAL x envrn),
```

which says that OPTIMIZE preserves the value of a form.

```
CORRECTNESS.OF.CODEGEN:
(FORMP x) -> (EXEC (REVERSE (CODEGEN x ins))
                   pds envrn)
                     =
             (PUSH (EVAL x envrn)
                   (EXEC (REVERSE ins)
                         pds envrn)),
```

which says that CODEGEN produces correct code when given a form.

```
FORMP.OPTIMIZE:
(FORMP x) -> (FORMP (OPTIMIZE x)),
```

which says OPTIMIZE produces a form if given one.

The theorem prover proves the two lemmas about OPTIMIZE without any help. However, in watching it try to prove the CORRECTNESS.OF.CODEGEN we saw that it needed to know that when EXEC executes the sequence (APPEND x y), then y is executed with the push-down stack produced by the execution of x. We called this the SEQUENTIAL.EXECUTION lemma:

```
(EXEC (APPEND x y) pds envrn)
                  =
(EXEC y (EXEC x pds envrn) envrn).
```

The system proves this lemma easily, by induction on the structure of x.

Once "cognizant" of this lemma, the theorem prover can prove the correctness of CODEGEN. This proof requires a rather subtle induction argument due to the fact that the compilation of the second argument position of a form takes place after the first argument has been compiled. The careful reader will note that the use of INS in CODEGEN (see Appendix A) is analogous to the use of Y in MC.FLATTEN (also in Appendix A): in both cases the functions recurse and pass down in the indicated argument position the value of another recursive call. Rather than present the details of the proof of CORRECTNESS.OF.CODEGEN we refer the reader to Appendix B where we present the system's proof of a simple theorem about MC.FLATTEN requiring a similar induction argument.

Once the system has proved CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.CODEGEN, and FORMP.OPTIMIZE it can prove the main theorem, CORRECTNESS.OF.OPTIMIZING.COMPILER, above, by rewriting:

```
(EXEC (REVERSE (CODEGEN (OPTIMIZE x) NIL))
      pds envrn)
```

to

```
(PUSH (EVAL (OPTIMIZE x) envrn) pds),
```

using the CORRECTNESS.OF.CODEGEN as a rewrite rule. [Note that in order to do so it must first establish (FORMP (OPTIMIZE x)). This is done by backwards chaining through FORMP.OPTIMIZE, picking up the (FORMP x) hypothesis from the main theorem]. Further simplification gives:

```
(FORMP x) -> (EVAL (OPTIMIZE x) envrn)
                        =
             (EVAL x envrn)
```

which is just CORRECTNESS.OF.OPTIMIZE, so the proof is complete. (Note that had we not already proved CORRECTNESS.OF.OPTIMIZE the system would have generated it and proved it here.)

The lemma CORRECTNESS.OF.CODEGEN is reminiscent of the correctness of a compiler for expressions presented by McCarthy and Painter in [13]. We have assumed that we have a push-down stack while they explicitly allocate registers. Diffie [14] and Milner and Weyhrauch [15] were able to proof check versions of this theorem with a considerable amount of user interaction. Cartwright [9] obtained a proof with less user assistance than [14] and [15]. Our automatic proof is distinguished from these proofs in that the proof of the lemma CORRECTNESS.OF.CODEGEN requires only that one lemma be proved ahead of time, does not require any guidance in doing induction, and handles the function CODEGEN, whose recursion is more efficient and more difficult to understand than the versions which employ APPEND. Our investigation of the subject of compiling expressions was inspired by the Buratall version of the theorem [10]. In fact, our first proofs employed functions akin to the LIT function

Burstall uses. After producing this proof and writing [12] we received a copy of Aubin's thesis [8]. In it he presents an automatic proof of a theorem extremely similar to our CORRECTNESS.OF.CODEGEN employing a single lemma analogous to SEQUENTIAL.EXECUTION.

Readers familiar with the old LISP theorem prover may be interested in knowing how that system fails to prove the correctness of our optimizing compiler. There are five basic inadequacies: its theory is too simple to permit a natural statement of the problem; it could not formulate the right induction for CODEGEN; its type handling is inadeaurte to simplify the induction arguments; it would run out of list space because it would not split often enough; and it could not have made sufficient use of lemmas.

IV HOW LEMMAS ARE USED

The new system uses lemmas in many ways; in this section we explain three of them.

A. Lemmas. for Rewriting

The most common use of axioms and lemmas is as rewrite rules during simplification. Of course, the simplifier uses function definitions as rewrite rules to "open up" a function call by replacing it with its definition (when the result is simpler in some sense). But more generally, the theorem prover interprets all axioms and lemmas as rewrite rules in the following way.

Given any formula (axiom or lemma) consider all of the sequents

H1 & H2 & ... & Hn ~> C

one can deduce from it by propositional calculus. We classify each sequent according to the form of C. If C is of the form (NOT u) then the rule can be used to rewrite any term which unifies with u to FALSE, provided the instantiated Hi can be established. If C is of the form (EQUAL u v) it can be used to rewrite u to v, again provided the instantiated Hi can be established. Otherwise, if C is Just u, where u is Boolean, then it can be used,to rewrite u to TRUE, under the same provision.

When the simplifier has decided to use a given rewrite it tries to establish the Hi by (recursively) simplifying them (to non-FALSE). This was illustrated above in the discussion of the proof of CORRECTNESS.OF.OPTIMIZING.COMPILER when the hypothesis (FORMP (OPTIMIZE x)) was established (i.e., rewritten to TRUE).

Care is taken to avoid infinite regression (e.g., repeated applications of a commutativity rewrite or "pumping" up a term with a rewrite like the contrapositive of FORMP.OPTIMIZE.) For example, we avoid the first problem by refusing to apply a rewrite of the form (EQUAL u v) when u and v ere variants unless the result is lexicographically less than the original formula. Thus, the lemmas

(PLUS i J) = (PLUS J i)

and

(PLUS i (PLUS j k)) = (PLUS j (PLUS i k)),

with the lexicographic restriction, cause nested PLUS expressions to be right associated with their arguments in (lexicographically) ascending order. (For example, (PLUS (PLUS y x) z) is normalized to (PLUS x (PLUS y z)) by the above described process.)

B. Lemmas for Generalizing

When trying to prove theorems by induction it is often necessary to prove a more general theorem than that given (where P is more general than Q if Q is an instantiation of P). Like the old LISP theorem prover, the system generalizes subterms common to both sides of an equality or implication by replacing the subterms with new variables. Unlike that theorem prover, the new system is sensitive to previously proved facts about the subterm.

Assume SORT is a list sorting function and ORDERED is a predicate that returns TRUE if the elements in its input list are in ascending order. Assume further that the system has proved that (SORT x) is a list of numbers when x is a list of numbers and that the system later has to prove:

```
(LIST.OF.NUMBERS x) &
    (ORDERED (SORT x)) & (NUMBERP i)
->
(ORDERED (MERGE i (SORT x))).
```

Then the system will generalize (SORT x) by replacing it with z, but will restrict z to being a list of numbers. That is, the system will adopt the new goal:

```
(LIST.OF.NUMBERS x) & (ORDERED z)
    & (NUMBERP i) & (LIST.OF.NUMBERS z)
->
(ORDERED (MERGE i z)).
```

(Note: (LIST.OF.NUMBERS x) would later be eliminated as irrelevant.) This use of lemmas allows the new system to avoid one of the most common failure modes of the old LISP theorem prover: generalizing a conjecture too much.

C. Lemmas for Inducting

As observed in Boyer and Moore [1], what makes an induction principle appropriate for a conjecture is whether it supplies inductive hypotheses about the recursive calls introduced when some of the functions in the induction conclusion are opened up.

1. Choosing an Appropriate Principle

For soundness, all of the induction hypotheses supplied must be instances of the theorem being proved, and, in the Generalized Principle of Induction [10], all the instances assumed must result from instantiating some n-tuple of variables in the conclusion with n-tuples of terms that are smaller (under the conditions governing the induction step) than the variable n-tuple in some well-founded partial order. The

trick, then, to finding an appropriate induction for a conjecture is to find a well-founded partial order on n-tuples and an n-tuple of variables in the conjecture, such that if one were to open up some of the function calls in the conjecture one would find subexpressions occurring that could be obtained by instantiating the conjecture itself with smaller n-tuples chosen from the ordering. Thus, one is lead naturally to the question: "What are some plausible well-founded orderings on n-tuples of variables that a given recursive function descends through in its recursion?"

To answer this question, our system imposes a basic responsibility upon the user to assume or to prove certain lemmas that we term "measure lemmas". Measure lemmas state that certain terms are less than other terms under well-founded orderings such as LESSP (on the non-negative integers.) Our induction facility then uses such lemmas at the time a recursive function is defined to determine some induction principles appropriate to that function. Of course, this induction mechanism is capable of examining all subsets of arguments to find appropriate principles. It also recognizes lexicographic orderings induced by existing orderings and chains together measure lemmas using transitivity (for example, to derive that (SUB1 (SUB1 x)) is less than x from the fact that (SUB1 x) is less than x.)

As an example of the use of such "measure lemmas" consider "big number" arithmetic. The problem is to represent and manipulate (in our case, add) integers that are larger than the word size of the host machine. The obvious solution is to represent them *as* sequences of digits in some chosen base, and to add them with the algorithm we all learned in the third grade. Two examples of the use of that (or similar) algorithms in computing are the "big number" arithmetic of MACLISP (where the base is 2) and the binary representation of integers on most machines (where the base is 2).

To state and prove the correctness of a big number addition algorithm one must define the mappings from integers to big numbers and back. Consider the first: To convert an integer i to a big number in base base, if i is less than base, then write down the digit i and stop, otherwise divide i by base, write down the remainder *as* the least significant digit, 8nd obtain the more significant digits by (recursively) converting the quotient to a big number in base base. This algorithm is embodied in the function POWER.REP (for "power series representation") in Appendix A, where big numbers are represented as lists of integers with least significant digit in the CAR.

Note that the algorithm above recurses on the quotient of i divided by base. What is the measure lemma that Justifies this recursion (or, analogously, the induction necessary to unwind it)? It is:

```
(i i 0) & (j i 0) & ((SUB1 j) i 0)
                  ->
      (QUOTIENT i J) < i ,
```

That is, the QUOTIENT of i divided by j is less than i if i is not 0, and j is neither 0 nor 1 (actually, in our untyped language, arithmetic on non-integers is legal and all non-integers *are* treated as though they were 1, thus (SUB1 j) *i* 0 is stronger than j *i* V, it implies that j is neither 1 nor a non-number).

Thus, if one wanted to prove by induction a theorem of the form P((POWER.REP i base)), and one were cognizant of the above lemma, a plausible induction to perform would be:

```
(i = 0) -> P((POWER.REP i base))
  &
(base = 0) -> P((P0WER.REP i base))
  &
((SUB! base) = 0) -> P((POWER.REP i base))
  &
((i i? 0) A (base i 0) 4 ((SUB1 base) i 0)
        h P((POWER.REP (QUOTIENT i base) base)))
                  ->
          PUPOWER.REP i base))
```

Note that the induction hypothesis is an instance of the theorem being proved, that the instantiation takes the tuple <i> into the smaller tuple <(QU0TIENT i b?se)>, according to the above measure lemma, and that the term (POWER.REP (QUOTIENT i base) base) appearing in the hypothesis will reappear when we open up the conclusion. Finally, observe that by employing proved measure lemmas we not only allow the theorem prover to be extended but we insure that it is done soundly. (Our first attempt to state the induction principle for QUOTIENT left out the J *i* 1 case and would have produced unsound inductions had it been either explicitly assumed or wired into the theorem prover program.)

The system has proved the measure lemma above. The proof involves a similar induction by DIFFERENCE (note in Appendix A how QUOTIENT itself recurses) which in turn was justified by a (proved) measure lemma stating that i-j is less than i under certain conditions. Once the QUOTIENT measure lemma has been proved the system, in accepting the definition of POWER.REP, will pre-process it to discover the plausible induction scheme above. That scheme is precisely the one the system uses to prove that the mapping back from big numbers to integers (POWER.EVAL in Appendix A) is the "inverse" (modulo non-numbers) of POWER.REP:

```
(EQUAL (POWER.EVAL (POWER.REP i base) base)
       (IF (NUMBERP i) i 1)).
```

The above lemma is crucial to the system's proof that a big number addition algorithm (BIG.PLUS in Appendix A) is correct:

```
(EQUAL (POWER.EVAL (BIG.PLUS (POWER.REP i base)
                             (POWER.REP j base)
                             0 base)
                   base)
       (PLUS i j)).
```

As a second example of a measure lemma, consider:

```
i < max -> max-(ADD1 i) < max -1
```

This lemma informs the system that it is sound to induct up by ADD1 to a maximum (e.g., that to prove P(x,y) inductively one may prove it when ~(x < y) (base case) and prove it when (x < y) assuming P(x+1,y)). In this ordering, <x+1,y> is less than <x,y>. This induction would be appropriate for a function which counted up.

2. Choosing the Right Instances

If one has a well-founded order on n-tuples then it can trivially be extended to a well-founded order on n+k-tuples, where the last k elements are simply irrelevant. Thus, once the system has found a plausible well-ordering that accounts for how n of the arguments change in recursion, it is free to throw in the remaining k arguments and let them change arbitrarily. One example of this occurs in the compiler proof. Recall that CODEGEN has two arguments, FORM and INS, and that during recursion it changes FORM by digging out one of its sub-expressions. However, while in one of its recursive calls INS (the instructions thus far laid down) is unchanged, in the other call INS is arbitrarily larger. This is acceptable since the behavior on FORM induces a well-founded order. Thus, the system knows that in a conjecture involving (CODEGEN form ins) an induction on the structure of form leaves it free to choose any instantiation it wants for ins. (For an example of such a choice in a simpler situation see Appendix B.) However, in a conjecture involving (CODEGEN form NIL) (where the INS argument is not a free variable) the system still knows it is sound to induct on form.

We originally tried to avoid such a deep and lemma driven analysis of the well-founded orderings used by functions by using subgoal induction [16], where, provided one assumes a function terminates, one can use the conditional structure of the function plus its recursive calls to define a well-founded order on the n-tuple of all of its arguments. However we found that in many theorems it was crucial to be able to induct on some subset of the arguments (knowing that they alone were sufficient to define an ordering) and that the conditional structure of most functions unnecessarily clutters inductions (which may seem like a mere inconvenience but often sets up too-restricted subgoals for subsequent inductions).

V REMARKS

We would like to reiterate that this paper reports work in progress (both on the features of our theory and the proof techniques). While the ourrent system is clearly not yet suitable as the theorem prover for practical program analysis, we *are* encouraged by our success at extending the theory of the our earlier LISP theorem prover without loss of proof-power. We believe we will be able to develop the system into a useful tool.

Appendix A

FUNCTION DEFINITIONS

Here we present the definitions of the functions involved in the optimizing compiler proof, the big number discussion, and the proof in Appendix B. Several axiomatically specified primitives are used. (NUMBERP x) is TRUE if x is a number and FALSE otherwise. (L1STP x) is TRUE if x is a CONS and FALSE otherwise. (PUSH x pds) returns a stack with x pushed onto pds. (TOP pds) returns the top-most element of a stack, and (POP pds) returns the popped stack. GETVALUE and APPLY are undefined (but APPLY is axiomatically specified to be numeric - a fact crucial to CORRECTNESS.OF.OPTIMIZE).

```
(APPEND (LAMBDA (X Y)
         (IF (LISTP X)
             (CONS (CAR X) (APPEND (CDR X) Y))
             Y)))

(BIG.PLUS
 (LAMBDA (X Y I BASE)
  (IF
   (LISTP X)
   (IF
    (LISTP Y)
    (CONS
     (REMAINDER (PLUS I (PLUS (CAR X) (CAR Y)))
                BASE)
     (BIG.PLUS
         (CDR X)
         (CDR Y)
         (QUOTIENT (PLUS I (PLUS (CAR X) (CAR Y)))
                   BASE)
         BASE))
    (BIG.PLUS1 X I BASE))
   (BIG.PLUS1 Y I BASE))))

(BIG.PLUS1
 (LAMBDA (L I BASE)
  (IF
   (LISTP L)
   (IF
    (EQUAL I 0)
    L
    (CONS
       (REMAINDER (PLUS (CAR L) I) BASE)
       (BIG.PLUS1 (CDR L)
                  (QUOTIENT (PLUS (CAR L) I) BASE)
                  BASE)))
   (CONS I NIL))))

(CODEGEN
 (LAMBDA (FORM INS)
```

```
(COMPILE
       (LAMBDA (FORM)
        (REVERSE (CODEGEN (OPTIMIZE FORM) NIL))))

(DIFFERENCE
       (LAMBDA (I J)
        (IF (EQUAL I 0)
            0
            (IF (EQUAL J 0)
                (IF (NUMBERP I) I 1)
                (DIFFERENCE (SUB1 I) (SUB1 J))))))

(EXEC
 (LAMBDA (PC PDS ENVRN)
  (IF
   (LISTP PC)
   (IF
    (LISTP (CAR PC))
    (IF
     (EQUAL (CAR (CAR PC)) (QUOTE PUSH1))
     (EXEC (CDR PC)
           (PUSH (CAR (CDR (CAR PC))) PDS)
           ENVRN)
     (EXEC
       (CDR PC)
       (PUSH (GETVALUE (CAR (CDR (CAR PC))) ENVRN)
             PDS)
       ENVRN))
    (EXEC (CDR PC)
          (PUSH (APPLY (CAR PC)
                       (TOP (POP PDS))
                       (TOP PDS))
                (POP (POP PDS)))
          ENVRN))
   PDS)))

(EVAL (LAMBDA (FORM ENVRN)
   (IF (LISTP FORM)
       (APPLY (CAR FORM)
              (EVAL (CAR (CDR FORM)) ENVRN)
              (EVAL (CAR (CDR (CDR FORM))) ENVRN))
       (IF (NUMBERP FORM)
           FORM
           (GETVALUE FORM ENVRN)))))

(FLATTEN (LAMBDA (X)
          (IF (LISTP X)
              (APPEND (FLATTEN (CAR X))
                      (FLATTEN (CDR X)))
              (CONS X NIL))))

(FORMP
  (LAMBDA (X)
   (IF (LISTP X)
       (IF (LISTP (CAR X))
           FALSE
           (IF (LISTP (CDR X))
               (IF (LISTP (CDR (CDR X)))
                   (IF (FORMP (CAR (CDR X)))
                       (FORMP (CAR (CDR (CDR X))))
                       FALSE)
                   FALSE))
       TRUE)))

(LESSP (LAMBDA (X Y)
        (IF (EQUAL X 0)
            (IF (EQUAL Y 0) FALSE TRUE)
            (IF (EQUAL Y 0)
                FALSE
                (LESSP (SUB1 X) (SUB1 Y))))))

(MC.FLATTEN
          (LAMBDA (X Y)
           (IF (LISTP X)
               (MC.FLATTEN (CAR X)
                           (MC.FLATTEN (CDR X) Y))
               (CONS X Y))))

(OPTIMIZE
 (LAMBDA (FORM)
  (IF
   (LISTP FORM)
   (IF
    (NUMBERP (OPTIMIZE (CAR (CDR FORM))))
    (IF
     (NUMBERP (OPTIMIZE (CAR (CDR (CDR FORM)))))
     (APPLY (CAR FORM)
            (OPTIMIZE (CAR (CDR FORM)))
            (OPTIMIZE (CAR (CDR (CDR FORM)))))
     (CONS
      (CAR FORM)
      (CONS
           (OPTIMIZE (CAR (CDR FORM)))
           (CONS (OPTIMIZE (CAR (CDR (CDR FORM))))
                 NIL))))
    (CONS
     (CAR FORM)
     (CONS (OPTIMIZE (CAR (CDR FORM)))
           (CONS (OPTIMIZE (CAR (CDR (CDR FORM))))
                 NIL))))
   FORM)))

(PLUS (LAMBDA (I J)
       (IF (EQUAL I 0)
           (IF (NUMBERP J) J 1)
           (ADD1 (PLUS (SUB1 I) J)))))

(POWER.EVAL
     (LAMBDA (L BASE)
      (IF (LISTP L)
          (PLUS (CAR L)
                (TIMES BASE
                       (POWER.EVAL (CDR L) BASE)))
          0)))

(POWER.REP
 (LAMBDA (I BASE)
  (IF
   (EQUAL I 0)
   NIL
   (IF
    (EQUAL BASE 0)
    (CONS I NIL)
    (IF
     (NUMBERP BASE)
     (IF
        (EQUAL BASE 1)
        (CONS I NIL)
        (CONS (REMAINDER I BASE)
              (POWER.REP (QUOTIENT I BASE) BASE)))
     (CONS I NIL))))))

(QUOTIENT
 (LAMBDA (I J)
  (IF (EQUAL J 0)
      0
      (IF (LESSP I J)
          0
          (ADD1 (QUOTIENT (DIFFERENCE I J) J))))))
```

```
(REMAINDER
       (LAMBDA (I J)
        (IF (EQUAL J 0)
            (IF (NUMBERP I) I 1)
            (IF (LESSP I J)
                (IF (NUMBERP I) I 1)
                (REMAINDER (DIFFERENCE I J) J)))))

(REVERSE (LAMBDA (X)
          (IF (LISTP X)
              (APPEND (REVERSE (CDR X))
                      (CONS (CAR X) NIL))
              NIL)))

(TIMES (LAMBDA (I J)
        (IF (EQUAL I 0)
            0
            (PLUS J (TIMES (SUB1 I) J)))))
```

Appendix B

THE FLATTEN.MC.FLATTEN THEOREM

This Appendix contains the machine's output during its proof of a relationship between two different tree flattening functions, FLATTEN and MC.FLATTEN (see Appendix A). The purpose is two-fold: To illustrate in 2 pages the theorem prover performing an induction similar to that done for CORRECTNESS.OF.CODEGEN (which takes 11 pages), and to exhibit the theorem prover's output.

PROOF OF THE "FLATTEN.MC.FLATTEN" LEMMA

The conjecture to be proved is:

```
(EQUAL (MC.FLATTEN X Y)
       (APPEND (FLATTEN X) Y))
```

Simplification produces:

```
*1.  (EQUAL (MC.FLATTEN X Y)
            (APPEND (FLATTEN X) Y)).
```

Give this the name *1. We'll try to prove it by induction.

There are 2 plausible inductions. These merge into one likely candidate induction. Induct on X (instantiating Y). This induction is justified by the CAR.LESSP and CDR.LESSP inequalities.

We must now prove the following 2 goals:

```
*1.i.  (IMPLIES (NOT (LISTP X))
                (EQUAL (MC.FLATTEN X Y)
                       (APPEND (FLATTEN X) Y)))
```

and

```
*1.ii.  (IMPLIES
         (AND
          (LISTP X)
          (EQUAL (MC.FLATTEN (CDR X) Y)
                 (APPEND (FLATTEN (CDR X)) Y))
          (EQUAL
              (MC.FLATTEN (CAR X)
                          (MC.FLATTEN (CDR X) Y))
              (APPEND (FLATTEN (CAR X))
                      (MC.FLATTEN (CDR X) Y))))
         (EQUAL (MC.FLATTEN X Y)
                (APPEND (FLATTEN X) Y))).
```

Simplification produces:

```
*1.iii.  (IMPLIES
          (AND
           (LISTP X)
           (EQUAL (MC.FLATTEN (CDR X) Y)
                  (APPEND (FLATTEN (CDR X)) Y))
           (EQUAL
               (MC.FLATTEN (CAR X)
                           (MC.FLATTEN (CDR X) Y))
               (APPEND (FLATTEN (CAR X))
                       (MC.FLATTEN (CDR X) Y))))
          (EQUAL
              (MC.FLATTEN (CAr X)
                          (MC.FLATTEN (CDR X) Y))
              (APPEND (APPEND (FLATTEN (CAR X))
                              (FLATTEN (CDR X)))
                      Y))).
```

Apply the lemma ASSOCIATIVITY.OF.APPEND to *1.iii. This produces:

```
*1.iv.  (IMPLIES
         (AND
          (LISTP X)
          (EQUAL (MC.FLATTEN (CDR X) Y)
                 (APPEND (FLATTEN (CDR X)) Y))
          (EQUAL
              (MC.FLATTEN (CAR X)
                          (MC.FLATTEN (CDR X) Y))
              (APPEND (FLATTEN (CAR X))
                      (MC.FLATTEN (CDR X) Y))))
         (EQUAL
           (MC.FLATTEN (CAR X)
                       (MC.FLATTEN (CDR X) Y))
           (APPEND (FLATTEN (CAR X))
                   (APPEND (FLATTEN (CDR X)) Y))))
```

Cross fertilize

```
    (MC.FLATTEN (CDR X) Y)
```

for

```
    (APPEND (FLATTEN (CDR X)) Y)
```

in *1.iv, and throw away the equality. This produces: TRUE.

That finishes the proof of *1.

Q.E.D.

```
Load average during proof:  .4605102
Elapsed time:  5.377 seconds
CPU time:  4.159 seconds
CONSes consumed:  4926
```

All of the commentary in the above proof was mechanically produced by the theorem prover in response to the user command to prove (EQUAL (MC.FLATTEN X Y) (APPEND (FLATTEN X) Y)). Several additional comments are in order.

The base case of the induction argument is formula *1.i and the induction step is •1.11. Note that in both cases the conclusions are the same (namely the theorem to be proved) and that in *1.ii the two induction hypotheses are instances of the theorem to be proved. One sends the tuple <X,Y> into <(*CDH* X),Y> and the other sends <X,Y> into <(CAR X),(MC.FLATTEN (CDR X) Y)>, both of which are smaller than <X,Y> according to the measure that compares only the first elements and uses the traditional ordering on list structures. The instantiation picked for Y in the second hypothesis was chosen because such a term would reappear when the conclusion was opened up.

Formula *1.i simplifies to TRUE and is thus proved. Formula *1.ii simplifies to *1.iii, which is proved by re-associating one of the APPEND terms (to produce *i.iv) and then doing a "cross-fertilization." (The result of "cross-fertilizing y for x in (IMPLIES (EQUAL x y) Q)" is Q with some occurrences of x replaced by y.)

If the associativity of APPEND had not been previously proved, the theorem prover would have automatically produced it (from *1.iv, by two cross-fertilizations, a generalization, and an elimination of an irrelevant hypothesis) and then proved it by induction.

REFERENCES

1. R. Boyer and J Strother Moore, "Proving Theorems about LISP Functions," MCM, Vol. *22,* No. 1, pp. 129-144 (1975).

2. J Strother Moore, "Computational Logic: Structure Sharing and Proof of Program Properties," Ph.D. thesis, University of Edinburgh (1973).

3. J Strother Moore, "Automatic Proof of the Correctness of a Binary Addition Algorithm," SIGART Newsletter. No. 52, pp. 13-14 (1975).

M. J Strother Moore, "Introducing Iteration into the Pure LISP Theorem Prover," IEEE Trans. Soft, Vol. 1, No. 3, pp. 328-338 (1975).

5. L. Robinson and K. Levitt, "Proof Techniques for Hierarchically Structured Prorams," CACM. Vol. 20, No. 4 pp. 271-283 (1977),

6. W. Bledsoe, "Splitting and Reduction Heuristics in Automatic Theorem Proving," Artificial Intelligence, Vol. 2, pp. 55-77 (197D.

7. W. Bledsoe, R. Boyer, and W. Henneman, "Computer Proofs of Limit Theorems," Artificial Intelligence, Vol. 3, PP. 27-60 (1972).

3. R. Aubin, "Mechanizing Structural Induction," Ph.D. Thesis, University of Edinburgh, 1976.

9. R. Cartwright, forthcoming Ph.D thesis, Computer Science Department, Stanford University, Stanford, California(1977).

10. R. M. Burstall, "Proving Properties of Programs by Structural Induction," The Computer Journal, Vol. 12, No. 1, pp. 41-48 (1969).

11. K. Clark and S-A. Tarnlund, "A First Order Theory of Data and Programs," Department of Computing and Control, Imperial College, London (1976).

12. R. Boyer and J Strother Moore, "A Computer Proof of the Correctness of a Simple Optimizing Compiler for Expressions," Technical Report 5, Contract N00014-75-C-0816, SRI Project 4079, Stanford Research Institute, Menlo Park, California (1977) (NTIS Number AD-A036 121/2WC).

13. J. McCarthy and J. Painter, "Correctness of a Compiler for Arithmetic Expressions," Mathematical Aspects of Computer Science, Vol. XIX, Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island, pp. 33-41 (1967).

14. W. Diffie, Unpublished disc file said to exist at Stanford Artificial Intelligence Laboratory by Richard Weyhrauch. (1972).

15. R. Milner and R. Weyhrauch, "Proving Compiler Correctness in a Mechanized Logic, Machine Intelligence 7, (eds. B. Meltzer and D. Michie) Edinburgh University Press, pp. 51-70 (1972).

16. J. Morris and B. Wegbreit, "Subgoal Induction," Xerox Palo Alto Research Center, CSL 75-6, Palo Alto, California (1975).