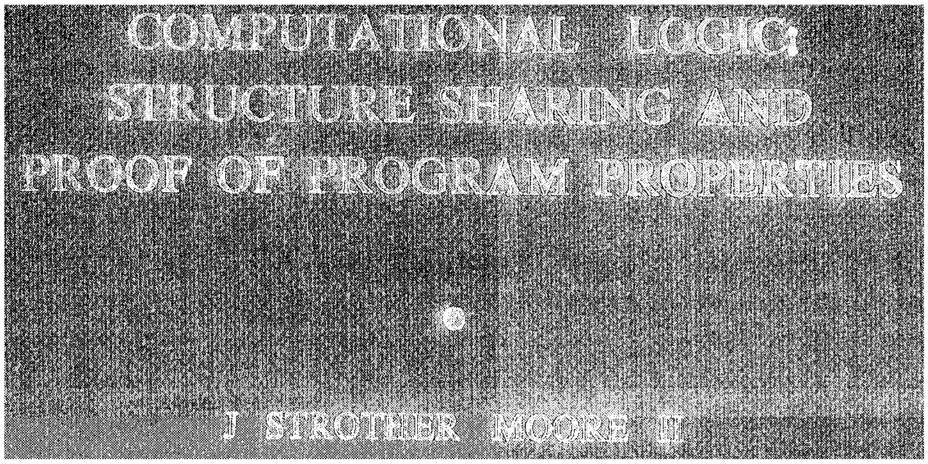


COMPUTATIONAL LOGIC: STRUCTURE SHARING AND PROOF OF PROGRAM PROPERTIES

J STROTHER MOORE II

COMPUTATIONAL LOGIC:

STRUCTURE SHARING AND PROOF OF PROGRAM PROPERTIES

J Strother Moore II

Doctor of Philosophy

University of Edinburgh

1973

ABSTRACT

This thesis describes the results of two studies in computational logic. The first concerns a very efficient method of implementing resolution theorem provers. The second concerns a non-resolution program which automatically proves many theorems about LISP functions, using structural induction.

In Part I, a method of representing clauses, called 'structure sharing', is presented. In this representation, terms are instantiated by binding their variables on a stack, or in a dictionary, and derived clauses are represented in terms of their parents. This allows the structure representing a clause to be used in different contexts without renaming its variables or copying it in any way. The amount of space required for a clause is (2 + n) 36-bit words, where n is the number of components in the unifying substitution made for the resolution or factor. This is independent of the number of literals in the clause and the depth of function nesting.

Several ways of making the unification algorithm more efficient are presented. These include a method of preprocessing the input terms so that the unifying substitution for derived terms can be discovered by a recursive look-up procedure. Techniques for naturally mixing computation and deduction are presented. The structure sharing implementation of SL-resolution is described in detail. The relationship between structure sharing and programming language implementations is discussed. Part I concludes with the presentation of a programming language, based on predicate calculus, with structure sharing as the natural implementation.

Part II of this thesis describes a program which automatically proves a wide variety of theorems about functions written in a subset of pure LISP. Features of this program include: The program is fully automatic, requiring no information from the user except the LISP definitions of the functions involved and the statement of the theorem to be proved. No inductive assertions are required from the user. The program uses structural induction when required, automatically generating its own induction formulas. All relationships in the theorem are expressed in terms of user defined LISP functions, rather than a second logical language. The system employs no built-in information about any non-primitive function. All properties required of any function involved in a proof are derived and established automatically. The program is capable of generalizing some theorems in order to prove them; in doing so, it often generates interesting lemmas. The program can write new, recursive LISP functions automatically in attempting to generalize a theorem. Finally, the program is very fast by theorem proving standards, requiring around 10 seconds per proof.

TABLE OF CONTENTS

PREFACE AND ACKNOWLEDGEMENTS

This thesis is composed of two independent parts. The first deals with an extremely efficient way to represent clauses in resolution theorem proving programs. The second deals with proving properties of LISP programs automatically, using structural induction. The link between these two topics is historical: The first study led to a representation of clauses that shares many features with techniques used in implementing programming languages. This allowed a resolution theorem prover to be written which efficiently interpreted a set of axioms which defined a programming language based on LISP. Since this program was a theorem prover (as well as an interpreter) it could be used to prove general theorems about some of the LISP functions defined. However, most interesting theorems required induction, and induction was not part of this theorem prover. This led to the second study, in which a completely new theorem prover was written. This system was not based on resolution, and incorporated induction as an automatic rule of inference. The only thing that survived the change was the LISP subset about which theorems were proved.

The two parts of this thesis are completely independent. Although the organization of both parts is the same, all chapter and section references are relative to the part containing the reference. Similarly, any reference to a formula with a certain name or number, refers to the so named formula of the section containing the reference.

During the work reported in both parts I received much help and encouragement from many people. Among these are: Woody Bledsoe, Alan Bundy, John Darlington, Michael Gordon, Pat Hayes, Bob Kowalski, Donald Kuehner, Robin Milner, Gordon Plotkin, Aaron Sloman, Rodney Topor, and Jophine van Vaalen.

In addition, I would like to thank Alan Robinson for his whole-hearted support of the structure sharing work, and Rod Burstall for his encouragement of the induction work.

I also want to give special thanks to Bernard Meltzer. He not only encouraged me constantly with his interest in the work being carried out, but he tolerated my unusual working habits and office hours. His leadership of the Department of Computational Logic has made it an exceptionally open and interesting group in which to work. This atmosphere in no small way contributed to the success of my endeavors.

The Science Research Council should be thanked for their financial support.

One of my supplementary thesis advisors, Robert S. Boyer, was of great help during the development of the ideas reported in this thesis. He made significant contributions to several of the concepts discussed. It was his conviction that an SL-resolution chain could be efficiently represented in terms of its parents and the unifying substitution. This led to the introduction of indices, and eventually, to general structure sharing. He originated the effort to automate induction by observing that it should be possible to discover what to induct upon by inspecting the definitions of the functions involved. He was also instrumental in the development of cross-fertilization and throwing the induction hypothesis away when it had been "used".

My gratitude to Bob Boyer, as an advisor, co-worker, and friend, is inexpressible. His experience with other formal systems and theorem provers was responsible for keeping the induction theorem prover firmly (and soundly) grounded in logic. This experience also led him to suggest the only strict rule regarding the manner in which research was conducted: No change in the program, no matter how trivial, was finalized until every theorem proved by the previous version was proved by the new version.

Finally, I wish to thank Bob Boyer, his wife, Anne, and my wife,Liz, for exceptional cooperation and understanding. They have all sacrificed much to a project which was of no benefit to any of them. Without these three people, this thesis would never have been written.

INTRODUCTION TO PART I

In implementing resolution theorem provers, one is immediately faced with the problem of how to represent clauses in the machine. The traditional approach is to use list structures. This suffers from the disadvantage of often requiring clauses to be copied before they can be used, and requiring the application of substitutions. This paper presents an alternative method, called structure sharing, which represents a derived clause by pointing to the parents and noting how the variables in them have been instantiated.

This method is very akin to the way programming languages are implemented. In particular, when a subroutine is called, the values of the formal arguments of the subroutine must be substituted into the body. In addition, care must be taken to prevent the confusion of variables in the calling program with those in the subroutine. The traditional theorem proving approach would be to copy the subroutine body and change the names of all of its variables. Then, copy it again, substituting the values for the variables. What is actually done is that the variables are kept straight by keeping track of how the recursion or function calling has been done, and they are "substituted for" by binding them in a stack or dictionary. The overhead involved is having to look up the values of variables whenever they are encountered, but there are ways that this can be done efficiently. The situation in theorem proving is somewhat more complex than it is in most programming languages, but an extension of the above methods actually works well.

Chapter 1 describes the basic ideas of structure sharing. Much of this chapter has been previously published (see Boyer and Moore, 1972). Chapter 1 alone is sufficient to obtain a complete understanding of structure sharing. The remaining chapters discuss modifications for efficiency and capabilities that structure sharing possesses which are not intuitively obvious.

Chapters 2 and 3 explain how the basic representation described in Chapter 1 can be reformulated for efficiency. These modifications can increase the speed of unification by an order of magnitude. These chapters are especially important to readers interested in implementing structure sharing.

Chapter 4 describes how structure sharing allows computation to be mixed arbitrarily with deduction. For example, it is both easy and efficient to introduce procedures in the implementation language which are automatically invoked when a term of a certain form is instantiated. Such a procedure might compute the value of the term.

Chapter 5 describes the implementation of SL-resolution. This is a particularly attractive linear resolution system. The implementation of it is very efficient and demonstrates how the basic ideas of structure sharing can be modified to deal efficiently with a restricted domain.

Finally, Chapter 6 presents a programming language based on predicate calculus, of which structure sharing is the natural implementation.

Throughout this paper, the programming language POP-2 is used to present algorithms. POP-2 has an ALGOL-like syntax and a LISP-like power. It was designed by R. Burstall and R. Popplestone of the

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh. A brief description of it is given below for readers unfamiliar with it.

All assignment, argument passing, and result returning is accomplished with a pushdown stack which is freely available to the user. To assign 5 to the identifier X, the syntax used is:

```
        5 -> X;                              (SETQ X 5)
```

(Equivalent LISP code will be exhibited in this introduction, where possible.)

Actually, the 5 in this context means "push 5 on the stack" and "-> X;" means "pop the (top of the) stack into the value cell for X". To push 5 on the stack and leave it there, one writes:

```
        5;
```

If there is something on the stack, it can be popped and assigned to X with:

```
        -> X;
```

Thus, to assign 5 to X and 4 to Y, either of the following may be used:

```
        5 -> X;                              (SETQ X 5)
        4 -> Y;                              (SETQ Y 4)
```

or:

```
        4; 5; -> X -> Y;
```

To assign 5 to the HD (CAR) of a list, X, one writes:

```
        5 -> HD(X);                          (RPLACA X 5)
```

To assign to the TL (CDR), an analogous statement is used:

```
        NIL -> TL(X);                        (RPLACD X NIL)
```

In fact, all assignment takes this form. The expression on the right of the "->" must just be an "updater" function, which takes as many things off the stack as it requires. If A is an array, it is accessed and updated in the expected way:

```
        A(I) + 3 -> A(J+2);
```

The stack is not protected against function entry and exit. In fact, arguments are passed via the stack, and results are returned on the stack. In the example above, the array A is actually a function which takes one argument off the stack, and leaves one result on the stack (when on the left-hand side of "->"). Thus, an equivalent way to get the Ith element of A onto the stack is:

```
        I; A();
```

That is, put I (the argument) on the stack, and then call A. This takes one thing off the stack, retrieves the relevant element, and puts it on the stack.

To form a list cell with 5 in the HD and NIL in the TL, and then assign it to X, the syntax is:

```
        CONS(5,NIL) -> X;                    (SETQ X (CONS 5 NIL))
```

What actually happens is that 5 and NIL are pushed, CONS is called and takes two things off the stack. CONS pushes a new list cell on the stack and exits, and the cell is popped into X. An equivalent piece of code is:

```
        5; NIL; CONS() -> X;
```

The syntax used in this paper is consistently that of the first CONS example above. This discussion has been to acquaint the reader with the stack.

To define a function in POP-2 with name FOO, formal arguments X and Y, and the locals Z and W, one writes:

```
FUNCTION FOO X Y;                   (DEFPROP FOO
VARS Z W;                                (LAMBDA (X Y)
   body                                  (PROG (Z W)
END;                                        body ))
                                         EXPR)
```

An example of a function which returns the sum and the product of its two arguments is:

```
FUNCTION SUMPROD X Y;
X + Y;
X * Y;
END;
```

The statement:

```
SUMPROD(4,5) -> PROD -> SUM;
```

assigns 20 to PROD and 9 to SUM.

The conditional statement in POP-2 has the form:

```
IF foo                              (COND (foo bar) (T mumble))
   THEN bar
   ELSE mumble CLOSE;
```

where foo, bar, and mumble are arbitrary POP-2 statements. The value of foo is tested against 0 rather than NIL. FALSE in POP-2 is 0; TRUE is 1.

The statement

```
IF foo THEN bar CLOSE;              (COND (foo bar))
```

allows ELSE clauses to be dropped if not needed.

```
IF foo
   THEN bar                         (COND (foo bar)
   ELSE IF mumble                         (mumble p)
      THEN p                              (T q))
      ELSE q CLOSE;
   CLOSE;
```

can be abbreviated to:

```
          IF foo
             THEN bar
             ELSEIF mumble
                THEN p
                ELSE q CLOSE;
```

to save the extra "CLOSE".

One very common feature in POP-2 programming is the use of the stack to return a truthvalue and one or more results. Throughout this paper the function ISBOUND is used. It takes two arguments and returns either TRUE and two additional results, or it returns FALSE. The common use of such a function is as follows:

```
          IF ISBOUND(V,I) THEN -> T -> J; CLOSE;
```

If ISBOUND(V,I) returns TRUE, two things are taken off the stack and assigned to T and J; otherwise, nothing is done. In either case, there is no net change with respect to the stack configuration after the above statement.

A function for testing whether X is in the list L is:

```
          FUNCTION MEMBER X L;            (DEFPROP
          IF L = NIL                         MEMBER
             THEN FALSE;                     (LAMBDA (X L)
             ELSEIF HD(L) = X                 (COND
                THEN TRUE;                     ((EQ L NIL) NIL)
                ELSE MEMBER(X,TL(L));          ((EQ (CAR L) X) T)
                CLOSE;                         (T (MEMBER X (CDR X)))))
         END;                                EXPR)
```

Iteration is available, but is used in only two forms in the programs exhibited here. One is the "LOOPIF" statement, which has the syntax of an IF without an ELSE:

```
          LOOPIF p THEN q CLOSE;
```

The meaning of the statement is that as long as p evaluates to non-FALSE, evaluate q and loop (back to the p test).

This is equivalent to:

```
     LOOP:                                   (PROG ( ... )
     IF p                                         .
        THEN                                      .
        q;                                        LOOP
        GOTO LOOP;                                (COND
        CLOSE;                                     (p  q (GO LOOP)))
                                                  .
                                                  . )
```

Because statements of the following form are used frequently:

```
     VARS L X;                               (PROG (L X ... )
     .                                            .
     .                                            .
     p -> L;                                      (SETQ L p)
     LOOPIF L /= NIL                              LOOP
        THEN                                      (COND
        HD(L) -> X;                                ((NOT (EQ L NIL))
        TL(L) -> L;                                 (SETQ X (CAR L))
        q;                                          (SETQ L (CDR L))
        CLOSE;                                      q
                                                    (GO LOOP)))
                                                  .
                                                  . )
```

the following abbreviation will be used:

```
     FOREACH X IN p;
     q;
     CLOSE;
```

POP-2 data structures include lists, arrays, and records. Pairs are an example of records. These are records with two components, FRONT and BACK. A pair is formed with the function CONSPAIR, which takes two arguments and constructs a record with two components. List cells are pairs, except that the TL of a list in POP-2 must be another pair, or NIL, while the BACK of a pair can be anything. If FOO is a function which accesses a component of a record, R, then accessing and assigning to that component is done with FOO as might be expected:

```
     FOO(R) + 5 -> FOO(R);
```

Only a very small subset of POP-2 is used in the programs exhibited. The syntax and language features provided are far richer than this introduction suggests. The POP-2 reference manual contains a complete description of the language.

CHAPTER 1 GENERAL STRUCTURE SHARING

1.1 Introduction

This paper is concerned with representing literals and clauses in computers. Lists provide the most obvious and natural representation of literals because function nesting imposes a tree-structure on literals. A list is also a reasonable representation of a set, in particular, of a clause. Lists however can consume large amounts of space and cause frequent garbage collections. This Chapter presents a representation of clauses and literals which is as natural as lists but far more compact. This economy is achieved by sharing the structure of the parents of a resolvent in the representation of the resolvent.

A clause is a set of literals; but throughout this paper the literals of a clause will be considered to be ordered.

Suppose C and D are clauses, and K is the ith literal of C, and L is the jth literal of D. Suppose further that the signs of K and L are opposite and that the substitution σ most generally unifies the atoms of K and L.

Under these hypotheses, C and D may be resolved on K and L to obtain the resolvent $R = ((C - \{K\}) \cup (D - \{L\}))\sigma$. The literals of R that come from C are considered to be "before" the literals that come from D. (Merging and factoring are ignored until Section 1.8.)

The tuple $< C,i,D,j,\sigma >$ contains sufficient information to enable the reconstruction of R. Therefore, in some sense, it represents R. At first sight it may not appear to be a very good representation of R, since it appears that the only way to use it would be to construct the list of literals and then apply σ to it.

However, a tuple like this is in fact very easy to use without constructing any lists or applying any substitutions.

1.2 Terms and Substitutions

To understand how to avoid applying substitutions, it is first necessary to understand the concept of the value of a term in the context of a substitution. First, an example:

```
(P X (F Y (G Z X)))
```

in the context of the substitution:

```
((Y.(F V W)) (Z.(G X U)) (U.(H X))),
```

is the term:

```
(P X (F (F V W) (G (G X (H X)) X))).
```

By a term is meant either a variable (e.g., X, Y, Z) or a list whose first member is a symbol (e.g., F, G, P, Q) and whose other members are terms.

By a substitution is meant a collection of pairs; the first member of each pair is a variable, and the second is a term. If (V.T) is a member of a substitution S, then V is said to be bound to T in S.

By the value of a term, TERM, in the context of a substitution,S, is meant the result of replacing each variable in TERM that is bound in S to a term, T, by the value of T in S.

These definitions are not those standard to the theorem proving literature. For example, there is no need to distinquish between predicate and function symbols (until one talks about what clauses represent). Also, there exists substitutions S such that some terms have no well-defined value in S. Precautions are taken never to generate such substitutions. Roughly speaking, a variable should not be bound twice or bound to a term whose value contains the variable.

It is possible to determine anything about the value of a term in the context of a substitution S without physically creating the value. The only thing one must do is: Whenever a variable, V, is encountered, S should be inspected to determine if V is bound in S to some term, TERM. If so, proceed as if TERM had been encountered instead of V.

Suppose that a substitution is represented by a list of pairs. Then the function ISBOUND, below, determines if its argument is bound in the global substitution S. If so, it leaves the term to which it is bound on the stack, along with TRUE, otherwise it returns FALSE:

```
FUNCTION ISBOUND V;
FOREACH PAIR IN S;
IF VAR = FRONT(PAIR) THEN BACK(PAIR); TRUE; EXIT;
CLOSE;
FALSE;
END;
```

If the function ISVAR returns TRUE if and only if its argument is a variable, then the following function determines whether some variable, V, occurs in the value of the term TERM, in the context of the global substitution S. Note that S is not applied to TERM.

```
FUNCTION OCCUR V TERM;
IF ISVAR(TERM)
   THEN
   IF ISBOUND(TERM)
      THEN -> TERM; OCCUR(V,TERM);
      ELSEIF V = TERM
         THEN TRUE;
         ELSE FALSE; CLOSE;
   ELSE
   FOREACH ARG IN TL(TERM);
   IF OCCUR(V,ARG) THEN TRUE; EXIT;
   CLOSE;
   FALSE;
   CLOSE;
END;
```

Notice that OCCUR checks to see if it has encountered a variable V' as TERM. If so, it checks to see if V' is bound. If V' is bound, it proceeds as if it had encountered the term to which V' is bound, by calling itself recursively on that term. If V' is not bound, it checks to see if V' is the variable V. If TERM is not a variable at all, it recursively checks each argument.

By avoiding the application of substitutions to terms it is possible to acheive a dramatic savings in space, which, of course, is paid for by looking up the bindings of variables. That this is worthwhile is demonstrated by the successful use of similar methods to "substitute" the values of formal parameters in programming languages, such as LISP and ALGOL.

1.3 Expressions and Bindings

The key to the representation of clauses presented here is the avoidance of physically creating the value of a term in the context of a substitution. This idea is at least as old as the first LISP. Terms and substitutions are not quite sufficient for the purpose, however, because it is often necessary to refer to different versions of a term. The concepts of an expression, a binding environment, and the value of an expression in a binding environment are introduced. First, some examples:

The value of the expression:

(P X (F Y (G Z X))),10

in the empty binding environment is the term:

$(P\ X_{10}\ (F\ Y_{10}\ (G\ Z_{10}\ X_{10})))$.

The value of the expression:

(P X (F Y (G Z X))),5

in the empty binding environment is the term:

$(P\ X_5\ (F\ Y_5\ (G\ Z_5\ X_5)))$.

Notice that the two values have no variable in common, despite the similarity of the two expressions.

The value of the expression:

(P X (F Y (G Z X))),5

in the binding environment:

(< Y,5,(F X Y),4 >

< Z,5,(G X U),5 >

< U,5,(H X),5 >),

is the term:

$(P\ X_5\ (F\ (F\ X_4\ Y_4)\ (G\ (G\ X_5\ (H\ X_5))\ X_5)))$.

By an index is meant a positive integer. By an expression is meant a term together with an index. If an expression is denoted by T,I, then T is a term, and I is an index.

By a binding is meant a four-tuple of the form < V,I,T,J >, where V is a variable, T is a term, and I and J are indices.

By a binding environment is meant a collection of bindings. If < V,I,T,J > is a member of the binding environment BNDEV, then V,I is said to be bound to T,J in BNDEV.

The value of an expression TERM,I in a binding environment, BNDEV, is the result of replacing each variable V in TERM by the value of V,I in BNDEV. If V,I is not bound in BNDEV, its value is the variable V_I (i.e., V subscript I). If V,I is bound in BNDEV to T,J, then its value is the value of T,J in BNDEV.

It is possible to determine anything about the value of an expression in a binding environment without physically creating the value.

Throughout this paper two routines will be used to facilitate the handling of binding environments. ISBOUND takes two arguments, V and I, and returns TRUE if V,I is bound in the global binding environment BNDEV. If it returns TRUE, the term and index to which V,I is bound are also put on the stack. If V,I is not bound, ISBOUND returns FALSE. Thus, if ISBOUND(V,I) returns TRUE, the statements:

```
-> TERM -> INDEX;
```

will assign the components of the expression to which V,I is bound to TERM and INDEX. The second routine is BIND. BIND(V,I,T,J) modifies the global binding environment, BNDEV, so that thereafter V,I is bound to T,J in BNDEV.

In the next three sections binding environments will be displayed as lists of bindings. This is done to help introduce the representation of clauses. The actual structure of a binding environment is made precise in Section 1.7. The only essential feature of a binding environment is that bindings can be discovered with ISBOUND and added with BIND.

Suppose it is necessary to determine whether some variable V_I occurs in the value of the expression TERM,J, in the binding environment BNDEV. The following recursive routine, OCCUR, does this:

```
FUNCTION OCCUR V I TERM J;
IF ISVAR(TERM)
   THEN
   IF ISBOUND(TERM,J)
      THEN -> TERM -> J; OCCUR(V,I,TERM,J);
      ELSEIF V = TERM AND I = J
         THEN TRUE;
         ELSE FALSE; CLOSE;
   ELSE
   FOREACH ARG IN TL(TERM);
   IF OCCUR(V,I,ARG,J) THEN TRUE; EXIT;
   CLOSE;
   FALSE;
   CLOSE;
END;
```

1.4 The Unification Algorithm

Suppose that VAL1 is the value of the expression TERM1,INDEX1 in the binding environment BNDEV. Suppose further that VAL2 is the value of the expression TERM2,INDEX2 in BNDEV. Finally, suppose that VAL is the most general common instance of VAL1 and VAL2 (if one exists). Then UNIFY(TERM1,INDEX1,TERM2,INDEX2) alters BNDEV so that the value of TERM1,INDEX1 in BNDEV and the value of TERM2,INDEX2 in BNDEV are both equal to VAL. If VAL1 and VAL2 have no common instance, the call to UNIFY returns FALSE. Otherwise, it returns TRUE. Like the definition of OCCUR in the previous Section, the routine below applies no substitutions.

```
FUNCTION UNIFY TERM1 INDEX1 TERM2 INDEX2;
LOOPIF ISVAR(TERM1) AND ISBOUND(TERM1,INDEX1)
   THEN -> TERM1 -> INDEX1; CLOSE;
LOOPIF ISVAR(TERM2) AND ISBOUND(TERM2,INDEX2)
   THEN -> TERM2 -> INDEX2; CLOSE;
IF TERM1 = TERM2 AND INDEX1 = INDEX2 THEN TRUE; EXIT;
IF ISVAR(TERM1)
   THEN
   IF OCCUR(TERM1,INDEX1,TERM2,INDEX2)
      THEN FALSE;
      ELSE BIND(TERM1,INDEX1,TERM2,INDEX2); TRUE; CLOSE;
ELSEIF ISVAR(TERM2)
   THEN
   IF OCCUR(TERM2,INDEX2,TERM1,INDEX1)
      THEN FALSE;
      ELSE BIND(TERM2,INDEX2,TERM1,INDEX1); TRUE; CLOSE;
ELSEIF HD(TERM1) = HD(TERM2)
   THEN
   FOREACH ARG1 ARG2 IN TL(TERM1) TL(TERM2);
   IF NOT(UNIFY(ARG1,INDEX1,ARG2,INDEX2)) THEN FALSE; EXIT;
   CLOSE;
   TRUE;
   ELSE FALSE; CLOSE;
END;
```

Here is an example of unification. Let TERM1 be (P X Y). Let TERM2 be (P (G X) Z). Let BNDEV be:

(< X,2,X,3 >

< Y,2,(F X Y),4 >

< Y,4,X,3 >

< Z,7,(F X Y),8 >

< X,8,X,7 >

< Y,8,(G Y),5 >).

The value of TERM1,2 in BNDEV is:

$$(P\ X_3\ (F\ X_4\ X_3)).$$

The value of TERM2,7 in BNDEV is:

$$(P\ (G\ X_7)\ (F\ X_7\ (G\ Y_5))).$$

After a call of UNIFY(TERM1,2,TERM2,7), BNDEV is:

$$\left.\begin{array}{l} (\langle X,7,Y,5 \rangle \\ \ \langle X,4,X,7 \rangle \\ \ \langle X,3,(G\ X),7 \rangle \end{array}\right\} \quad \text{added by UNIFY}$$

$$\left.\begin{array}{l} \ \langle X,2,X,3 \rangle \\ \ \langle Y,2,(F\ X\ Y),4 \rangle \\ \ \langle Y,4,X,3 \rangle \\ \ \langle Z,7,(F\ X\ Y),8 \rangle \\ \ \langle X,8,X,7 \rangle \\ \ \langle Y,8,(G\ Y),5 \rangle). \end{array}\right\} \quad \text{old bindings in BNDEV}$$

The value of TERM1,2 in the new BNDEV is:

$$(P\ (G\ Y_5)\ (F\ Y_5\ (G\ Y_5))).$$

The value of TERM2,7 in the new BNDEV is:

$$(P\ (G\ Y_5)\ (F\ Y_5\ (G\ Y_5))).$$

1.5 Incrementing Indices and Standardizing Expressions Apart

Let T be the term:

(Q (F X (A)) (G Y Z)).

The value of the expression T,5 in the binding environment BNDEV1:

(< X,5,(G Y Z),5 >

< Z,5,(F (A) U),6 >

< U,6,X,3 >),

is the term:

(Q (F (G Y_5 (F (A) X_3)) (A)) (G Y_5 (F (A) X_3))).

The value of the expression T,11 in the binding environment BNDEV2:

(< X,11,(G Y Z),11 >

< Z,11,(F (A) U),12 >

< U,12,X,9 >),

is:

(Q (F (G Y_{11}(F (A) X_9)) (A)) (G Y_{11}(F (A) X_9))).

Notice that the value of T,5 in BNDEV1 is a variant of the value of T,11 in BNDEV2; furthermore, the values have no common variables. The values are 'standardized apart.' Notice also that BNDEV2 is just the result of incrementing every index in BNDEV1 by 6.

Suppose that T is a term, BNDEV1 is a binding environment, and BNDEV2 is obtained by adding increment INC to every index in BNDEV1. Then the value of T,J in BNDEV1 is a variant of the value of T,J+INC in BNDEV2. If INC is greater than or equal to the largest index in BNDEV1,then the two values have no variables in common.

1.6 Resolving Clauses Using Expressions and Bindings

This Section describes by example how expressions, incrementing indices, and the unification procedure work together in resolution. In this Section a simple representation of clauses is used, namely, a list of expressions in a binding environment. After one resolution has been performed using this representation the main point of this chapter is presented.

The list C1:

```
((+ (Q Y Y)),2  (+ (P X Y)),2  (- (P X (F Y Z))),4)
```

in the binding environment BNDEV1:

```
(< X,2,X,3 >
 < Y,2,(F X Y),4 >
 < Y,4,X,3 >
 < Z,4,(F X Y),2 >),
```

represents the clause CLAUSE1:

$$((+\ (Q\ (F\ X_4\ X_3)\ (F\ X_4\ X_3)))$$
$$(+\ (P\ X_3\ (F\ X_4\ X_3)))$$
$$(-\ (P\ X_4\ (F\ X_3\ (F\ X_3\ (F\ X_4\ X_3))))))$$

in an obvious way.

Similarly, the list C2:

```
((- (Q X Y)),1  (- (P (G X) Z)),3  (+ (R X (F X Y))),4),
```

in the binding environment BNDEV2:

```
(< Z,3,(F X Y),4 >
 < X,4,X,3 >
 < Y,4,(G Y),1 >
 < Y,2,(G Z),3 >),
```

represents the clause CLAUSE2:

$$((-\ (Q\ X_1\ Y_1))$$
$$(-\ (P\ (G\ X_3)\ (F\ X_3\ (G\ Y_1))))$$
$$(+\ (R\ X_3\ (F\ X_3\ (G\ Y_1))))).$$

To resolve CLAUSE1 and CLAUSE2 on their second literals, they must be standardized apart. This is done by incrementing all of the indices in C2 and BNDEV2 by 4 (the maximum index in C1 or BNDEV1). Call the results C2' and BNDEV2', respectively. C2' in the binding environment BNDEV2' represents the clause, CLAUSE2':

((- (Q X_5 Y_5))
 (- (P (G X_7) (F X_7 (G Y_5))))
 (+ (R X_7 (F X_7 (G Y_5))))).

The expression representing the second literal of CLAUSE1 is:

(+ (P X Y)),2.

The expression representing the second literal of CLAUSE2' is:

(- (P (G X) Z)),7.

Since their signs are opposite, UNIFY is called as follows:

UNIFY((P X Y),2,(P (G X) Z),7).

Of course, UNIFY requires the global binding environment BNDEV to be set. In this case, it is initialized to BNDEV1 $\cup$ BNDEV2'. The call to UNIFY returns TRUE and side effects BNDEV so that it is:

$$\begin{array}{ll}
\left.\begin{array}{l} (< X,7,Y,5 > \\ < X,4,X,7 > \\ < X,3,(G\ X),7 > \end{array}\right\} & \text{added by UNIFY} \\
\left.\begin{array}{l} < X,2,X,3 > \\ < Y,2,(F\ X\ Y),4 > \\ < Y,4,X,3 > \\ < Z,4,(F\ X\ Y),2 > \end{array}\right\} & \text{BNDEV1} \\
\left.\begin{array}{l} < Z,7,(F\ X\ Y),8 > \\ < X,8,X,7 > \\ < Y,8,(G\ Y),5 > \\ < Y,6,(G\ Z),7 >). \end{array}\right\} & \text{BNDEV2'}
\end{array}$$

The resolvent, RESOLVENT, could be represented by a list of expressions, R, in the binding environment BNDEV, above. R is obtained by appending C1 and C2' after deleting their second literals:

```
((+ (Q Y Y)),2
 (- (P X (F Y Z))),4
 (- (Q X Y)),5
 (+ (R X (F X Y))),8).
```

In the binding environment BNDEV, R represents:

$$((+\ (Q\ (F\ Y_5\ (G\ Y_5))\ (F\ Y_5\ (G\ Y_5))))$$
$$(-\ (P\ Y_5\ (F\ (G\ Y_5)\ (F\ (G\ Y_5)\ (F\ Y_5\ (G\ Y_5))))))$$
$$(-\ (Q\ X_5\ Y_5))$$
$$(+\ (R\ Y_5\ (F\ Y_5\ (G\ Y_5)))))$$

It should be obvious that it is exceedingly wasteful to physically create the lists C2' and R, and the binding environments BNDEV2' and BNDEV, given their definitions in terms of C1, BNDEV1, C2, and BNDEV2. The representation does not actually create any of C2', R, BNDEV2', or BNDEV. A clause is represented so that the following may be easily retrieved: (1) the expression for the nth literal, and (2) the binding of V,I (if it is bound). Under the hypothesis that (1) and (2) can be retrieved for CLAUSE1 and CLAUSE2, the tuple used to represent RESOLVENT contains precisely enough information to allow the retrieval of (1) and (2) for RESOLVENT.

Assume inductively that it is possible to retrieve the expression for the nth literal of either parent, CLAUSE1 or CLAUSE2, of a resolvent RESOLVENT. The expression for the nth literal of RESOLVENT is either the expression for some literal of CLAUSE1 or the expression for some literal of CLAUSE2 with the index incremented by MI, the maximum index of CLAUSE1. Exactly which literal of the parent is a function of n, the number of literals in CLAUSE1, and the numbers of the literals resolved upon in CLAUSE1 and CLAUSE2. The following diagram should make it clear how to compute the position of the expression in the parent. The algorithm is given in the next Section. In the example below, L3 and K2 are resolved upon.

$$\begin{array}{c} \overbrace{\text{L1 L2 L3 L4 L5}}^{\text{CLAUSE1}}\ \overbrace{\text{K1 K2 K3 K4 K5 K6}}^{\text{CLAUSE2}} \\ \underbrace{\text{R1 R2}\ \phantom{\text{L3}}\ \text{R3 R4 R5}\ \phantom{\text{K2}}\ \text{R6 R7 R8 R9}}_{\text{RESOLVENT}} \end{array}$$

Assume, again inductively, that it is possible to determine whether V,I is bound to T,J in the binding environment of either parent. If the representation of a resolvent includes the bindings made by the unification for the resolution, it is possible to determine if V,I is bound in the binding environment of the resolvent. In particular, V,I is bound to T,J if and only if:

V,I is bound to T,J in the bindings added by the unification for RESOLVENT, or

I $\leq$ MI and V,I is bound to T,J in the binding environment of the left-hand parent, CLAUSE1, or

I $>$ MI and V,I-MI is bound to T,J-MI in the binding environment of the right-hand parent, CLAUSE2.

If the expressions and binding environments for input clauses can be computed, then they can be computed for a derived clause if the following information is included in the record of such a clause:

(1) the record representing the left parent,

(2) the number of the literal resolved upon in the left parent,

(3) the record representing the right parent,

(4) the number of the literal resolved upon in the right parent,

(5) the number of literals in the resolvent,

(6) the maximum index in the resolvent,

(7) the bindings added during the unification for the resolvent.

This is precisely the information in the clause representation described in detail in the next Section.

1.7 The Details of the Representation

By a clause record is meant either an input record or a resolvent record. An input record is a list of literals. A literal has a sign, which may be + or -, and an atomic formula, which is a term in the sense of Section 1.2. Here are two input records:

```
((+ (P X (F Y))))
((- (Q X X)) (+ (P (F X) Y)) (+ (R (A) Z))).
```

If IP is an input record, then LITCNT(IP) is the length of IP, and MAXINDEX(IP) is 1.

By a resolvent record, R, is meant a structure with seven components. The components are described below and the name of the function which accesses each component is given in parentheses:

(1) a clause record (LEFTPAR),

(2) an integer (LEFTLIT),

(3) a clause record (RIGHTPAR),

(4) an integer (RIGHTLIT),

(5) an integer (LITCNT),

(6) an integer (MAXINDEX),

(7) a list of bindings (BINDINGS).

The function CONSCLAUSE takes seven arguments and constructs a clause record.

By a binding is meant a record with four components. The components are as described below.

(1) a variable (VCOMP),

(2) an integer (VICOMP),

(3) a term (TCOMP),

(4) an integer (TICOMP).

The function CONSBIND takes four arguments and constructs a binding record.

The components of the resolvent record represent those objects listed at the end of Section 1.6. The components of a binding record represent those listed in Section 1.3.

To obtain the expression for the Kth literal of a clause record, CL, GETLIT is used. GETLIT returns the term and index for the appropriate expression. NTHMEMB returns the nth member of a list.

```
FUNCTION GETLIT K CL;
VARS TERM INDEX;
IF ISINPUT(CL)
   THEN 1; NTHMEMB(K,CL);
ELSEIF K < LEFTLIT(CL)
   THEN GETLIT(K,LEFTPAR(CL));
ELSEIF K < LITCNT(LEFTPAR(CL))
   THEN GETLIT(K+1,LEFTPAR(CL));
ELSEIF K < LITCNT(LEFTPAR(CL)) - 1 + RIGHTLIT(CL)
   THEN
   GETLIT(K-LITCNT(LEFTPAR(CL))+1,RIGHTPAR(CL))
      -> TERM -> INDEX;
   INDEX + MAXINDEX(LEFTPAR(CL));
   TERM;
   ELSE
   GETLIT(K-LITCNT(LEFTPAR(CL))+2,RIGHTPAR(CL))
      -> TERM -> INDEX;
   INDEX + MAXINDEX(LEFTPAR(CL));
   TERM;
   CLOSE;
END;
```

To determine if V,I is bound in the global binding environment of the clause record BNDEV, ISBOUND is used. As defined below, ISBOUND has the same calling structure and results as previously defined. However, the definition below is in terms of the actual representation of binding environments.

```
FUNCTION ISBOUND VAR INDEX;
ISBOUND1(VAR,INDEX,0,BNDEV);
END;
```

ISBOUND1 is a recursive function which inspects the bindings in BNDEV and then branches to the left or right parent if a binding is not found:

```
FUNCTION ISBOUND1 VAR INDEX INCRMT BNDEV;
IF ISINPUT(BNDEV) THEN FALSE; EXIT;
FOREACH B IN BINDINGS(BNDEV);
IF VAR = VCOMP(B) AND INDEX = VICOMP(B)
   THEN
   TICOMP(B) + INCRMT;
   TCOMP(B);
   TRUE;
   EXIT;
CLOSE;
IF INDEX =< MAXINDEX(LEFTPAR(BNDEV))
   THEN
   ISBOUND1(VAR,INDEX,INCRMT,LEFTPAR(BNDEV));
   ELSE
   ISBOUND1(VAR,
            INDEX - MAXINDEX(LEFTPAR(BNDEV)),
            INCRMT + MAXINDEX(LEFTPAR(BNDEV)),
            RIGHTPAR(BNDEV));
   CLOSE;
END;
```

To add a binding of V,I to T,J to BNDEV, BIND is used:

```
FUNCTION BIND V I T J;
CONS(CONSBIND(V,I,T,J),BINDINGS(BNDEV)) -> BINDINGS(BNDEV);
END;
```

To resolve two clause records, CL1 and CL2 on their Ith and Jth literals respectively, RESOLVE is used:

```
FUNCTION RESOLVE CL1 I CL2 J;
VARS LEFTTERM LEFTI RIGHTTERM RIGHTI;
GETLIT(I,CL1) -> LEFTTERM -> LEFTI;
GETLIT(J,CL2) -> RIGHTTERM -> RIGHTI;
CONSCLAUSE(CL1,I,
           CL2,J,
           LITCNT(CL1) + LITCNT(CL2) - 2,
           MAXINDEX(CL1) + MAXINDEX(CL2),
           NIL) -> BNDEV;
IF HD(LEFTTERM) /= HD(RIGHTTERM) AND
   UNIFY(HL(TL(LEFTTERM)),LEFTI,
         HD(TL(RIGHTTERM)),RIGHTI + MAXINDEX(CL1))
   THEN BNDEV; TRUE;
   ELSE FALSE; CLOSE;
END;
```

If the unification is successful, BNDEV is the clause record of the resolvent (UNIFY modifies its BINDINGS component with the function BIND). In this case, it is left on the stack along with TRUE. If the unification fails, FALSE is returned. The functions UNIFY and OCCUR are exactly as in Sections 1.4 and 1.3, except that they use the definitions of BIND and ISBOUND given in this Section.

The most time consuming function in RESOLVE is the unification step. In particular, it should be noted that the clauses are standardized apart entirely by incrementing indices. Except for the exploration on the two literals by UNIFY, the work involved in creating the resolvent is independent of the complexity of the two parents. No copying is done, and no substitutions are applied.

Figure 1 exhibits a derivation involving three input clauses and four resolvents. The clause records, delimited by '<' and '>', are presented, along with the clauses they represent.

The clauses labelled C1, C2, and C3 are input clauses. The four remaining clauses are generated by RESOLVE as follows:

```
C4 = RESOLVE(C1,1,C2,2);

C5 = RESOLVE(C1,1,C4,2);
```

C1: ((- (P X (F Y))) (- (Q (F X) (F Y))))

C2: ((+ (P X Y)) (+ (P Y X)) (+ (Q X Y)))

C3: ((+ (Q X (F X))) (+ (Q X Y)))

C4: < C1,1,C2,2,3,2,(< X,2,(F Y),1 > < X,1,Y,2 >) >

C5: < C1,1,C4,2,3,3,(< Y,3,(F Y),1 > < X,1,(F Y),2 >) >

C6: < C4,1,C3,1,3,3,(< Y,1,(F X),1 > < X,3,(F X),1 >) >

C7: < C5,1,C6,3,4,6,(< Y,6,(F Y),1 > < Y,5,(F Y),2 >) >

The records C4 through C7 represent the clauses:

C4: ((- (Q (F Y_2) (F Y_1))) (+ (P (F Y_1) Y_2)) (+ (Q (F Y_1) Y_2)))

C5: ((- (Q (F (F Y_2)) (F Y_1))) (- (Q (F (F Y_1)) (F Y_2)))
(+ (Q (F Y_2) (F Y_1))))

C6: ((+ (P (F (F Y_2)) Y_2)) (+ (Q (F (F Y_2)) Y_2)) (+ (Q (F Y_2) Y_3)))

C7: ((- (Q (F (F Y_1)) (F Y_2))) (+ (Q (F Y_2) (F Y_1)))
(+ (P (F (F (F Y_2))) (F Y_2))) (+ (Q (F (F (F Y_2))) (F Y_2))))

The tree representing the derivation of C7 is:

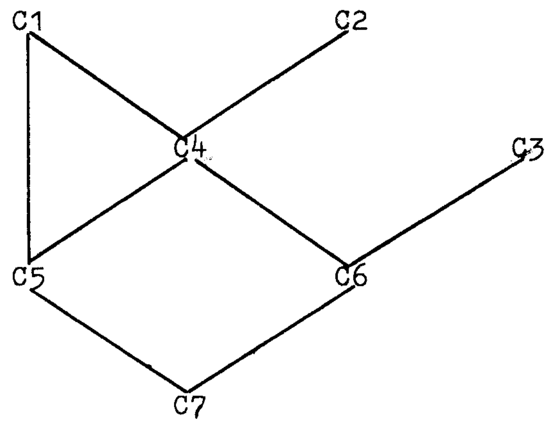


Figure 1. At the top are three input clauses and four clause records. The order of the components in the records is the same as that given in this Section.

```
C6 = RESOLVE(C4,1,C3,1);
C7 = RESOLVE(C5,1,C6,3);
```

It is useful to trace the descent of the third literal of C2 through the tree. In C2 it is represented by the expression T,1, where T is:

(+ (Q X Y)).

In the binding environment of C2, this expression has the value:

(+ (Q X_1 Y_1)).

The descendant of this literal in clause C4 is the third literal of that clause. There the expression is T,2, and has the value:

(+ (Q (F Y_1) Y_2)).

In C5 the term has index 3 and represents the third literal:

(+ (Q (F Y_2) (F Y_1))).

Meanwhile, back at C4, the expression T,2 descends along a different branch to become the second literal of C6, as the expression T,2:

(+ (Q (F (F Y_2)) Y_2)).

When C5 and C6 are resolved to form C7, the term T descends from both parents. With index 3 it represents the second literal, and with index 5 it represents the fourth literal. In C7, T,3 has value:

(+ (Q (F Y_2) (F Y_1))),

and T,5 has value:

(+ (Q (F (F (F Y_2))) (F Y_2))).

In obtaining the value of T,3, bindings in C5 are used, while for T,5, bindings in C6 are used. In both cases, bindings in C4 are used.

1.8 Notes on the General Representation

Merging and factoring have been ignored up to this point. They present no difficulty however. In order to represent a merge, or factor, some literal has to be deleted and some substitution applied. This can be done in structure sharing by providing a dummy input clause with one literal. In order to merge two literals in a clause, C, the unifying bindings are produced and a resolvent record is built which uses the dummy input clause as one of the parents, and C as the other. The BINDINGS are those produced by UNIFY. The literal in C resolved upon, LEFTLIT, is the one to be deleted.

Since many components of resolvent records are small integers, it is possible to pack these so that the record requires very few machine words. An efficient POP-2 implementation of the representation described above requires 7+2n 24-bit words per clause, where n is the number of bindings produced by UNIFY. A machine code implementation of the general representation on a 36-bit word machine would require 2+n words per clause.

Statistics have been obtained comparing this representation to two others, namely the most obvious list representation, and the most compact character array imaginable. The latter is extremely slow to use since most of one's time is spent parsing. Assuming a 36-bit word machine is used, structure sharing is 10 times more compact that character arrays (at 5 characters per word) and 50 to 100 times more compact than lists (at 1 cons per word). This is based on 3000 randomly generated clauses.

Using the simple implementation described in this chapter, the program requires 160 milliseconds per unification (on the average). The average longest branch searched by ISBOUND was 8.3. The average function nesting depth was 5. For comparison purposes with other machines, it should be pointed out that these timings were done in POP-2 on an ICL 4130, where the time required to execute the POP-2 expression '1 + 2' in a compiled function body is 160 microseconds.

CHAPTER 2 MODIFICATIONS OF UNIFY AND OCCUR FOR EFFICIENCY

2.1 UNIFY and its Variants

There are four very similar procedures commonly found in resolution theorem provers. These are used for deciding whether some term, TERM1, unifies with a term TERM2, is identical to TERM2, is a variant of TERM2, or subsumes TERM2.

It is obvious that two terms are identical if and only if they are unified by the empty substitution. Therefore, if binding is prohibited, the code for UNIFY can be made to perform the task of IDENT. However, binding can be prevented by simply redefining the function OCCUR so that it always returns TRUE. Since OCCUR must return FALSE if a binding is to be made, no binding will occur. If UNIFY must make a binding to succeed, it will call OCCUR, which will return TRUE, causing UNIFY to fail.

In structure sharing it is possible to redefine OCCUR in two other ways to cause UNIFY to perform the tasks of VARIANT and TERMSUBSUME, provided TERM1 and TERM2 are standardized apart.

Let TERM1 be the value of T1,I1 (in some binding environment), and let TERM2 be the value of T2,I2 (in the same binding environment). Assume that all the indices of T1,I1 are less than those of T2,I2. This can be arranged by incrementing indices provided the two terms are to be standardized apart.

TERM1 is a variant of TERM2 if they are identical up to a one-to-one renaming of the variables of TERM1 onto those of TERM2. This means they are variants if and only if they are unified by a substitution which only binds variables from T1,I1 to variables from T2,I2, such that no variable from T2,I2 appears as the term components of more than one binding in the substitution. Since a redefinition of OCCUR can use the index of a variable or term to determine whether the variable or term has come from T1,I1 or T2,I2, such a function can be used to guarantee that if UNIFY succeeds, the substitution produced has the above form.

Assume that BNDEV has been set as if the clauses from which T1,I1 and T2,I2 come were to be resolved. Then OCCUR can allow a binding (that is, it can return FALSE) if and only if the index of the variable is less than or equal to MAXINDEX(LEFTPAR(BNDEV)), the term is a variable, and the term is not already the term component of any binding in BINDINGS(BNDEV). A suitable definition of OCCUR for VARIANT testing is:

```
FUNCTION VAROCCUR V I TERM J;
IF ISVAR(TERM) AND I =< MAXINDEX(LEFTPAR(BNDEV))
   THEN
   FOREACH B IN BINDINGS(BNDEV);
   IF TERM = TCOMP(B) AND J = TICOMP(B)
      THEN TRUE; EXIT;
   CLOSE;
   FALSE;
   ELSE TRUE; CLOSE;
END;
```

Thus, T1,I1 from clause C1 is a variant of T2,I2 from clause C2 if when BNDEV is set as it would be to resolve C1 and C2 and OCCUR is redefined as VAROCCUR above, UNIFY(T1,I1,T2,I2) returns TRUE.

If TERM1 and TERM2 are standardized apart, TERM1 subsumes TERM2 if and only if they can be unified by a substitution that binds only variables from TERM1. Provided the indices of TERM1 are less than those of TERM2, a suitable definition of OCCUR for TERMSUBSUME is:

```
FUNCTION SUBOCCUR V I TERM J;
IF I =< MAXINDEX(LEFTPAR(BNDEV))
   THEN FALSE;
   ELSE TRUE; CLOSE;
END;
```

Thus, the procedure for determining whether one term subsumes another is exactly like the above description for VARIANT, except that the definition of SUBOCCUR is used as OCCUR in UNIFY.

Being able to use the code for UNIFY in four distinct ways to provide four very basic resolution functions is a significant economy of space. It is possible because the index of an expression indicates the origin of the term in the derivation of the clause represented by BNDEV.

2.2 Preprocessing Input Terms

One of the most remarkable facts about structure sharing is that it is possible to preprocess the input terms in such a way that they can be thrown away before the search begins. The only information needed about any two terms in a pure resolution theorem prover is that needed to compute their most general unifier and to determine whether a given variable occurs in them. If one determines these things for the input terms, then one can do it for derived terms.

Suppose that T1,1 and T2,2 in the empty binding environment are unified by the bindings in the list BINDLIST. Further, suppose that T1,I1 and T2,I2 in the binding environment BNDEV are to be unified. Let BINDLIST' be the result of replacing every occurrence of 1 as an index in BINDLIST by I1, and every such occurrence of 2 by I2. Then T1,I1 and T2,I2 are identical in any binding environment if and only if V,I and TERM,J are identical in that environment, for every binding

< V,I,TERM,J > in BINDLIST'. Thus, to unify T1,I1 and T2,I2 in BNDEV it is sufficient to treat BINDLIST' as a list of pairs of expressions which must be unified.

An example should make this clear. Let T1,I1 be:

    (P (F (G X)) (G Y)),7

and let T2,I2 be:

    (P (F Y) Y),8.

The list of bindings that unify T1,1 and T2,2 in the empty binding environment is BINDLIST:

    (< Y,1,X,1 > < Y,2,(G X),1 >).

Thus, BINDLIST' is:

    (< Y,7,X,7 > < Y,8,(G X),7 >).

Then T1,I1 and T2,I2 in BNDEV may be unified by unifying Y,7 and X,7 and Y,8 and (G X),7 in BNDEV. The point is that T1 and T2 do not have to be reexplored to determine how to unify them: a table-lookup of the original unifying substitution may be used.

In the example above, Y,8 may be bound in BNDEV to, say, (G (G Z)),6. Then when UNIFY(Y,8,(G X),7) is called, it will discover the binding and recursively try to unify (G (G Z)),6 and (G X),7. It is clear that if the initial unifying substitution for (G (G Z)),1 and (G X),2 were available in a table, the procedure could be applied again.

Except for OCCUR checking, the only thing needed to unify any two T1,I1 and T2,I2 is the initial unifier.

If all the input terms are replaced by integers that denote their positions in a table of unifications, and if the bindings in the table are expressed in terms of these integers, then the terms themselves may be thrown away as far as unification is concerned.

The details are as follows. Associate with each of the N distinct terms in the input set an integer, called the position of the term. Let UNITABLE be an N x N array. UNITABLE(P1,P2) contains either the message that the term with position P1, index 1 does not unify with the term with position P2, index 2 in the empty binding environment, or a list of bindings unifying those two expressions. The bindings are of the form < V,I,P,J >, where V is a variable, P is a position (the position of the term to which V,I is bound), and I and J are each 1 or 2.

Once UNITABLE is produced, the clauses can be replaced by lists of positive and negative integers. The absolute value of such an integer gives the position of the appropriate atom in the table, and the arithmetic sign gives the logical sign of the literal. The literal access function, GETLIT, is unchanged by this radical modification of input clauses. The function RESOLVE must check that the arithmetic signs of the two "literals" are opposite, and then call UNIFY on their absolute values and the normal indices.

UNIFY is just as it was before, except that if neither term is a variable, it retrieves the contents of UNITABLE(TERM1,TERM2). If this is a list, UNIFY recurses on each pair of expressions in the list, replacing the dummy indices 1 and 2 by the values of INDEX1 and INDEX2. The code is given below. RDUMMYIND is a function which replaces a dummy index by the proper one:

```
FUNCTION RDUMMYIND DUMMY;
IF DUMMY = 1
   THEN INDEX1;
   ELSE INDEX2; CLOSE;
END;
```

```
FUNCTION TABUNIFY TERM1 INDEX1 TERM2 INDEX2;
LOOPIF ISVAR(TERM1) AND ISBOUND(TERM1,INDEX1)
   THEN -> TERM1 -> INDEX1; CLOSE;
LOOPIF ISVAR(TERM2) AND ISBOUND(TERM2,INDEX2)
   THEN -> TERM2 -> INDEX2; CLOSE;
IF TERM1 = TERM2 AND INDEX1 = INDEX2 THEN TRUE; CLOSE;
IF ISVAR(TERM1)
   THEN
   IF TABOCCUR(TERM1,INDEX1,TERM2,INDEX2)
      THEN FALSE;
      ELSE BIND(TERM1,INDEX1,TERM2,INDEX2); TRUE; CLOSE;
ELSEIF ISVAR(TERM2)
   THEN
   IF TABOCCUR(TERM2,INDEX2,TERM1,INDEX1)
      THEN FALSE;
      ELSE BIND(TERM2,INDEX2,TERM1,INDEX1); TRUE; CLOSE;
ELSEIF UNITABLE(TERM1,TERM2) = "FAIL"
   THEN FALSE;
   ELSE
   FOREACH B IN UNITABLE(TERM1,TERM2);
   IF NOT(TABUNIFY(VCOMP(B),RDUMMYIND(VICOMP(B)),
                   TCOMP(B),RDUMMYIND(TICOMP(B))))
      THEN FALSE; EXIT;
   CLOSE;
   TRUE;
   CLOSE;
END;
```

The preprocessing thus frees the unification algorithm from exploring the terms. Instead it jumps immediately to the position where they may not match and attempts to unify the critical terms. Furthermore, if two nested terms fail to unify initially, UNIFY does not recurse to discover it. Each term is initially explored and thereafter only the information gained by that exploration is needed.

However, it is still necessary to be able to determine whether a variable occurs in the term with position P. This has a solution precisely parallel to the table driven unification. Each term is preprocessed, and a list is formed containing the variables that occur in it. This list is stored at position P of the linear array of length N, OCCTABLE.

OCCUR then becomes a function with four arguments as before, V,I and P,J. However, now P can either be a variable or a position. If it is a variable, OCCUR is just as it was before. If it is a position, the contents of OCCTABLE(P) are retrieved and OCCUR recurses on each variable in that list. Once again, the preprocessing has prevented the exploration of the terms.

The input terms can be completely discarded given the existence of UNITABLE and OCCTABLE. The only difficulty is that it is then impossible to print anything meaningful. However, UNIFY and OCCUR are several times faster, depending upon the function nesting in the input set and the degree of instantiation of the "typical" derived terms being unified.

Below is an example of a set of input clauses preprocessed in the manner described.

Let three of the clauses be:

```
C1:  ((- (P (F X) X)) (+ (Q (G Y X)))),
C2:  ((+ (P (A) (F X))) (+ (P (F (A))Y)) (- (Q (G (F X) (A))))),
C3:  ((- (P (A) X)) (+ (Q (G X X)))).
```

The association of terms to numbers (as assigned by the most obvious recursive function) is:

1	(P (F X) X)
2	(F X)
3	(Q (G Y X))
4	(G Y X)
5	(P (A) (F X))
6	(A)
7	(P (F (A)) Y)

8	(F (A))
9	(Q (G (F X) (A)))
10	(G (F X) (A))
11	(P (A) X)
12	(Q (G X X))
13	(G X X)

This association is used to print terms.

The OCCTABLE has the following entries:

I	OCCTABLE(I)
1	(X)
2	(X)
3	(X Y)
4	(X Y)
5	(X)
6	NIL
7	(Y)
8	(X)
9	(X)
10	(X)
11	(X)
12	(X)
13	(X)

Finally, the non-trivial elements of UNITABLE are as below. All other elements are the failure message.

I	J	UNITABLE(I,J)
1	1	(< X,1,X,2 >)
1	7	(< Y,2,6,2 > < X,1,6,2 >)
2	2	(< X,1,X,2 >)
2	8	(< X,1,6,2 >)
3	3	(< X,1,X,2 > < Y,1,Y,2 >)
3	9	(< X,1,6,2 > < Y,1,2,2 >)
3	12	(< X,1,X,2 > < Y,1,X,2 >)
4	4	(< X,1,X,2 > < Y,1,Y,2 >)
4	10	(< X,1,6,2 > < Y,1,2,2 >)
4	13	(< X,1,X,2 > < Y,1,X,2 >)
5	5	(< X,1,X,2 >)
5	11	(< X,2,2,1 >)
6	6	NIL
7	1	(< Y,1,6,1 > < X,2,6,1 >)
7	7	(< Y,1,Y,2 >)
8	2	(< X,2,6,1 >)
8	8	NIL
9	3	(< X,2,6,1 > < Y,2,2,1 >)
9	9	(< X,1,X,2 >)
10	4	(< X,2,6,1 > < Y,2,2,1 >)
10	10	(< X,1,X,2 >)
11	5	(< X,1,2,2 >)
11	11	(< X,1,X,2 >)
12	3	(< Y,2,X,2 > < X,1,Y,2 >)
12	12	(< X,1,X,2 >)
13	4	(< Y,2,X,2 > < X,1,Y,2 >)
13	13	(< X,1,X,2 >)

The three input clauses become:

C1: (-1 3),

C2: (5 7 -9),

C3: (-11 12).

To unify the first literal of C1 and the second literal of C2, with indices 5 and 8 respectively, TABUNIFY(1,5,7,8) is called. This retrieves UNITABLE(1,7):

(< Y,2,6,2 > < X,1,6,2 >),

and produces two recursive calls:

TABUNIFY(Y,8,6,8), and (if that succeeds)

TABUNIFY(X,5,6,8).

These two recursive calls use the same procedure.

Note that TABUNIFY does not need to know that term 1 is:

(P (F X) X),

and term 7 is:

(P (F (A)) Y).

It only needs to know that to unify 1 and 7, the X must be unified with (A) and Y with (A). The original inspection of 1 and 7 detected that the P's and F's were matched.

The idea of preprocessing the input terms in this way was developed jointly by several members of the Department of Computational Logic, including Bob Kowalski, Bob Boyer, and Ed Wilson. In addition, J. van Vaalen, of Mathematisch Centrum, Amsterdam, contributed to this work during her visit to the Department of Machine Intelligence.

2.3 Avoiding Unnecessary OCCUR Checks

It is possible to significantly reduce the number of calls to OCCUR during a resolution unification by the following observation. If two clauses are being resolved, they are standardized apart. Thus, a variable from the left-hand parent will not occur in a term from the right-hand parent unless during this unification, there has been a binding of a variable from the right to a term from the left. A similar statement holds for left-to-right bindings. Once again, in structure sharing, this condition is easy to check.

All bindings are made by the function BIND, which can use the indices to determine the origins of the variable and term. Let BIND be modified so that when the variable's index is greater than MAXINDEX(LEFTPAR(BNDEV)) and the term's index is less than or equal to MAXINDEX(LEFTPAR(BNDEV)), a right-to-left flag is set. This signifies that there has been a binding of a variable from the right to a term from the left.

When OCCUR is called it can return FALSE if the variable's index is less than or equal to MAXINDEX(LEFTPAR(BNDEV)), the term's index is greater than MAXINDEX(LEFTPAR(BNDEV)), and the right-to-left flag is not set. Under these conditions, the variable (which is from the left) cannot possibly occur in the term (from the right). If the indices are not in the proper relationship, or the flag is set, the traditional (table-driven) OCCUR is called.

Similar modifications are made for the symmetric, left-to-right, case. The result is that the first OCCUR check in a resolution is always avoided, and the second is avoided 25 percent of the time.

P. Roussel, of the Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Marseille, has also made these observations about OCCUR.

CHAPTER 3 MODIFICATIONS OF ISBOUND AND BIND FOR EFFICIENCY

3.1 The VALUE Array

A significant amount of time is spent in ISBOUND. However, if a clause is going to be used extensively, for example, involved in several resolutions, it can be processed so as to make the time spent in ISBOUND insignificant. This is achieved by collecting all of the bindings made in the derivation of the clause, and storing them (temporarily) in an array that allows direct access to them. This two dimensional array is called the VALUE array. Its columns are labelled by variables and its rows by indices. Conceptually, VALUE(V,I) contains either the atom UNBOUND, meaning V,I is not bound in the binding environment,BNDEV, of the clause, or it contains an expression, T,J, to which V,I is bound in BNDEV. Actually, it will contain the conventional four-tuple binding record. VCOMP and VICOMP of this record will always be V and I. The TCOMP component will contain either the message UNBOUND, or a term, and TICOMP will contain the index of the term.

The following is the VALUE array as it would conceptually look if it were loaded with the bindings in the binding environment of C7 in Figure 1.

	X	Y	Z
1	(F Y),2	UNBOUND	UNBOUND
2	Y,3	UNBOUND	UNBOUND
3	(F Y),2	(F Y),1	UNBOUND
4	Y,5	(F X),4	UNBOUND
5	(F Y),4	(F Y),2	UNBOUND
6	(F X),4	(F Y),1	UNBOUND
7	UNBOUND	UNBOUND	UNBOUND
.	.	.	.

Note that when the bindings of RIGHTPAR of C7 are loaded, the indices are uniformly incremented by MAXINDEX of LEFTPAR, since this is how ISBOUND would interpret them from C7.

The recursive function for loading the VALUE array, given a clause CL and an increment INC is just:

```
FUNCTION LOAD CL INC;
IF ISINPUT(CL) THEN EXIT;
FOREACH B IN BINDINGS(CL);
TCOMP(B) -> TCOMP(VALUE(VCOMP(B),VICOMP(B)+INC));
TICOMP(B) + INC -> TICOMP(VALUE(VCOMP(B),VICOMP(B)+INC));
CLOSE;
LOAD(LEFTPAR(CL),INC);
LOAD(RIGHTPAR(CL),INC+MAXINDEX(LEFTPAR(CL)));
END;
```

To LOAD a clause into VALUE is considerably faster than checking ISBOUND just once for each variable in its representation, since each node in the tree is inspected exactly once. However, after the binding environment has been loaded, ISBOUND is an insignificant array access:

```
FUNCTION VALISBOUND V I;
IF TCOMP(VALUE(V,I)) = UNBOUND
   THEN FALSE;
   ELSE TICOMP(VALUE(V,I)); TCOMP(VALUE(V,I)); TRUE; CLOSE;
END;
```

If VALISBOUND is used in place of ISBOUND in UNIFY, then BIND must be altered to modify the VALUE array rather than the BINDINGS list of BNDEV. Of course, the only thing that VALBIND (the VALUE array version of BIND) does is assign the term and index to the appropriate components of the VALUE cell determined by the variable and its index. The previous definitions of UNIFY and OCCUR will work as usual, but considerably faster.

When the binding environment currently in the VALUE array is of no more interest (a search strategic decision) VALUE must be unloaded in preparation for loading a new clause. The function UNLOAD does this; this function has the same structure as LOAD, except that it assigns UNBOUND to TCOMP of the appropriate VALUE cells.

The VALUE array is particularly suited to depth first search. Assume that the resolvent, C3, of C1 and C2 is to be the left parent of the next resolvent, and assume that C1 is already loaded in preparation for resolution with C2. If C2 is loaded into the VALUE array with its index incremented by MAXINDEX(C1), then the array will contain the proper environment for the unification step for forming C3. If VALISBOUND and VALBIND are used, then when the unification is completed, VALUE contains the binding environment for C3. Thus, C3 is now properly loaded as the left parent of the next resolvent.

As outlined above, it is impossible to recover the list of bindings produced by unification. To do this, VALBIND must also keep a record of the VALUE cells it modifies. The most natural way to do this is to keep pointers to the cells on a pushdown stack. Let BINDSTACK be a pushdown stack with stack pointer BINDPTR. Let PUSH be a function which pushes its first argument on the stack pointed to by its second argument. Then VALBIND may be defined as:

```
FUNCTION VALBIND V I TERM J;
TERM -> TCOMP(VALUE(V,I));
J -> TICOMP(VALUE(V,I));
PUSH(VALUE(V,I),BINDSTACK);
END;
```

To recover the substitution produced by UNIFY and written into the VALUE array by VALBIND, the value of BINDPTR should be saved before the unification is initiated. After the unification is completed, the bindings on BINDSTACK between the top of the stack (BINDPTR) and the old value of

BINDPTR are the appropriate bindings. Note that if the bindings are collected to store in the BINDINGS component of a clause record, the bind records must be copied (since to share the bind records with the VALUE array would be disastrous). The function GETBINDS collects the bindings and returns them in a list.

```
FUNCTION GETBINDS OLDPTR;
VARS SAVEPTR BINDLIST;
BINDPTR -> SAVEPTR;
NIL -> BINDLIST;
LOOPIF BINDPTR /= OLDPTR
   THEN
   CONS(COPYBINDS(POP(BINDSTACK)),BINDLIST) -> BINDLIST;
   CLOSE;
SAVEPTR -> BINDPTR;
BINDLIST;
END;
```

An additional complication of using the VALUE array is that if a unification fails, the BINDSTACK must be used to reset the VALUE cells modified before the failure occurred. The function UNBIND does this; it merely writes UNBOUND into the TCOMP component of each cell on BINDSTACK between the current pointer and the saved one. The stack pointer is restored to its previous configuration.

```
FUNCTION UNBIND OLDPTR;
LOOPIF BINDPTR /= OLDPTR
   THEN
   UNBOUND -> TCOMP(POP(BINDSTACK));
   CLOSE;
END;
```

BINDSTACK and UNBIND can be used to write very efficient and elegant recursive programs for depth-first search. Below is a description of a recursive factoring routine, which factors the factors as they are produced. Suppose that the clause being factored is initially loaded into the VALUE array.

Upon entry to FACTOR, the current value of BINDPTR is saved in OLDBINDPTR. Then a two pointer search through the pairs of literals in the clause is started. Unification is attempted on those literals with with the same sign. If a unification succeeds, the VALUE array will have been modified by the UNIFY so that it is properly loaded for the factor. The program then forms the record for the factor, deleting the appropriate literal, stores the clause produced on an answer list, and then calls itself recursively on the factor. When the recursive call returns, the routine uses UNBIND between BINDPTR and OLDBINDPTR to restore VALUE to its configuration before the unification. The two pointer search is then continued. When finished, the routine exits with VALUE in precisely the configuration it was in upon entry.

A very similar recursive routine for subsumption checking can be written using TERMSUBSUME, defined in Chapter 2.

The VALUE array can be created once and for all at the beginning of a theorem proving session, and can be used to make the processing of any clause (or set of clauses) very efficient. It also prevents the allocation of space during unifications which fail. A theorem proving program running with structure sharing and a VALUE array requires additional space only when the decision to keep a clause has been made. No space is required during the unification process itself. This makes the garbage collection behaviour of the program very efficient. Finally, the VALUE array means that the binding information in a clause record can be efficiently packed without sacrificing processing time in using the clause.

CHAPTER 4 COMBINING COMPUTATION AND DEDUCTION

4.1 Restrictor Functions

It is sometimes useful to attach recommendations to a clause as to how it should be used. A good example of this occurs in the Blind Hand Problem by Robin Popplestone (Michie et al., 1972). One of the axioms in this problem is:

```
((- (AT X Y Z)) (+ (AT X Y (DO (LETGO) Z)))),
```

which means that if X is at place Y in situation Z, then it is still at Y in the situation that results if a LETGO action is performed in situation Z. This is a "frame axiom" used to ensure that a LETGO does not change the positions of objects. However, it has the property that it can be resolved with itself, or several similar axioms, to produce literals of the form:

```
(+ (AT X Y (DO (LETGO) (DO (LETGO) (DO ...)))))
```

involving a situation term of little or no use.

While this problem can be solved by reformulating the axioms, it is very natural to simply attach some restrictions on the use of the axioms. A particularly simple approach is to require that Z above never be bound to a term of the form (DO (LETGO) ...).

Of course, this restriction cannot be checked merely when the axiom is used. For instance, when the axiom is resolved with, Z may get bound to some free variable, Z', and later, Z' may be bound to (DO (LETGO) ...). Another possibility is for Z to be bound to (DO Z' ...) when the axiom is used, and for Z' to be bound to (LETGO) later. The point is that these restrictions must be checked throughout the derivation.

Thus, if Z is bound to any term, T, with free variables, Z_i, then the relationship between Z and the Z_i must be known to the routine that enforces restrictions, since binding the Z_i affects the value of Z. Furthermore, to detect whether some variable, Z, has a restriction on it, the input clause from which Z descended must be known, since presumably the restriction will be found there. Clearly, structure sharing is called for.

Assume throughout the following that the VALUE array is being used. Let a restriction be expressed in terms of a function which is called on two arguments, a term and an index, and returns TRUE or FALSE. Assume that it is possible to optionally associate with any variable in an input clause a restrictor function which returns TRUE if and only if the current value of the expression represented by the arguments is a permitted binding of the variable concerned. (Restrictor functions usually inspect the VALUE array and are associated with variables via an association list stored with the input clause.)

Let LINKARRAY be an array of the dimensions of VALUE. It will be used to hold a possibly empty list of variables whose values are affected by the value of each V,I in the array.

Let RESTARRAY be an array of the dimensions of VALUE. It will be used to hold a possibly empty list of restrictions on V,I.

The function LOAD must be modified so that when it loads an expression T,J with variables Z_1, ..., Z_n in T, into VALUE(V,I), it adds V,I to the lists in LINKARRAY(Z_i,J) for i = 1 to n. When LOAD encounters an input clause it retrieves the association list of restrictor functions for variables in that clause and for each pair (V.f) on the list, it adds f to RESTARRAY(V,INC+1), where INC is the increment at which the clause is being LOADed. f is supposed to be a function of two arguments, which returns TRUE if and only if its arguments represent an expression whose value is a permitted instantiation of the variable V.

The definition of LOAD is:

```
FUNCTION LOAD CL INC;
IF ISINPUT(CL)
   THEN
   FOREACH PAIR IN RESTFNS(CL);
   CONS(BACK(PAIR),RESTARRAY(FRONT(PAIR),INC+1))
      -> RESTARRAY(FRONT(PAIR),INC+1);
   CLOSE;
   EXIT;
FOREACH B IN BINDINGS(CL);
TCOMP(B) -> TCOMP(VALUE(VCOMP(B),VICOMP(B)+INC));
TICOMP(B)+INC -> TICOMP(VALUE(VCOMP(B),VICOMP(B)+INC));
ADDLINKS(VCOMP(B),VICOMP(B)+INC,TCOMP(B),TICOMP(B)+INC);
CLOSE;
LOAD(LEFTPAR(CL),INC);
LOAD(RIGHTPAR(CL),INC+MAXINDEX(LEFTPAR(CL)));
END;
```

where the function ADDLINKS adds a new set of links to LINKARRAY:

```
FUNCTION ADDLINKS V I T J;
IS ISVAR(T)
   THEN
   CONS(CONSPAIR(V,I),LINKARRAY(T,J)) -> LINKARRAY(T,J);
   ELSE
   FOREACH ARG IN TL(T);
   ADDLINKS(V,I,ARG,J);
   CLOSE;
   CLOSE;
END;
```

Thus, if V,I is on the list in LINKARRAY(Z,J), then the value of V,I is affected by that of Z,J. That is, when the value of Z,J is instantiated, then so is that of V,I. If FOO is a function on the list in RESTARRAY(V,I), then FOO is a restrictor function which must "approve" of the value of V,I. In particular, FOO must approve of any changes in the value of V,I produced by instantiation.

For example, assume that Y,8 is bound to (DO Y (NOW)),7, with Y,7 free, and assume that FOO is a restrictor function on Y,8 that prohibits it from being bound to a term with value of the form (DO (LETGO) ...). LINKARRAY(Y,7) will contain Y,8 (and perhaps other variable expressions), and RESTARRAY(Y,8) will contain FOO (and perhaps other restrictor functions).

If Y,7 is to be bound to (LETGO), it must be "satisfied" with that value. That is, all of the restrictions in RESTARRAY(Y,7) must be met when that binding is in force. Furthermore, all of the variables in LINKARRAY(Y,7) must also be satisfied with their new values. In particular, even if the restrictions on Y,7 permit it to be bound to (LETGO), Y,8 must be satisfied with that binding in force. So the restrictions in RESTARRAY(Y,8) must be checked and FOO will be found.

FOO will discover that the new value of Y,8 is (DO (LETGO) (NOW)), and will return FALSE. Thus, Y,8 is not satisfied with its value, so neither is Y,7. The result is that Y,7 cannot be bound to (LETGO).

The above sketch should suggest the necessary modifications to BIND and UNIFY. When BIND is called on V,I and T,J, it should temporarily insert T,J into VALUE(V,I) and then call the new function ISSATISFIED on V,I to see if V,I is satisfied with its value.

The function ISSATISFIED takes a variable and index as arguments and retrieves the list of restrictions on the variable expression. If all of the restrictions are met, it retrieves the list of linked variables and recursively determines whether each of them is satisfied. If so, it returns TRUE. Otherwise, it returns FALSE.

```
FUNCTION ISSATISFIED V I;
FOREACH RESTFN IN RESTARRAY(V,I);
IF RESTFN(V,I) = FALSE
   THEN FALSE; EXIT;
CLOSE;
FOREACH PAIR IN LINKARRAY(V,I);
IF ISSATISFIED(FRONT(PAIR),BACK(PAIR)) = FALSE
   THEN FALSE; EXIT;
CLOSE;
TRUE;
END;
```

If BIND finds that ISSATISFIED returns FALSE, it must remove T,J from VALUE(V,I) and return FALSE to UNIFY. Otherwise, BIND must update LINKARRAY for the new binding (adding V,I to the lists associated with the variables in T), and return TRUE to UNIFY.

Thus, the definition of BIND is now:

```
FUNCTION BIND V I T J;
VARS OLDPTR;
BINDPTR -> OLDPTR;
T -> TCOMP(VALUE(V,I));
J -> TICOMP(VALUE(V,I));
PUSH(VALUE(V,I),BINDSTACK);
IF ISSATISFIED(V,I)
   THEN ADDLINKS(V,I,T,J); TRUE;
   ELSE UNBIND(OLDPTR); FALSE; CLOSE;
END;
```

UNIFY must treat BIND as a predicate with side-effects. If BIND returns TRUE, then the VALUE array has been updated to cause the binding, and all restrictions have been met, so UNIFY can return TRUE. If BIND returns FALSE, the VALUE array is unchanged because some restriction was violated by the desired binding, and UNIFY must return FALSE, just as if V,I had occurred in T,J.

While involving substantial modifications of several routines, the inclusion of restrictor functions is extremely natural to structure sharing. In addition, the axiom writer is given great flexibility. In particular, he is able to impose restrictions on the use of the axioms in a way that is powerful and intuitive. Furthermore, the restrictions are implemented as computational steps rather than derivational ones.

The Blind Hand Problem, referred to earlier, provides a good example of the use of restrictor functions. The refutation of this set of axioms is known for its difficulty. An SL-resolution theorem prover, using a depth-first search with depth limit 17 (the depth of the refutation) generated 30000 clauses without finding a proof. However, very intuitive restrictor functions permitted the same program to find a proof after about 30 clauses. This is the only known, fully automatic, first order proof of this theorem.

The restrictor functions used were as follows. If Z occurred in an input term of the form (DO (LETGO) Z), the restriction on Z was that it could not be bound to a term which had a (LETGO) action in it unless that (LETGO) was succeeded by a (PICKUP) in the term. Thus, the action (LETGO) was not allowed unless a (PICKUP) had occurred after the last (LETGO) in the situation. A similar restriction on (PICKUP) was imposed, and (GO ...) was allowed only if some other action had occurred after the last (GO ...). These functions are trivial list processing (shared structure processing) functions which inspect expressions. Their effects were dramatic.

The mechanisms set up to handle restrictor functions have a more general application, described in the next Section.

4.2 Automatic Evaluation

In many kinds of problems axiomatized for automatic theorem provers, it is necessary to include axioms for arithmetic, even though they are only to be used to compute trivial arithmetic facts, such as:

(GT (ADD 3 4) 5).

Arithmetic is one of many domains where it is easier to compute the valuesof certain expressions than it is to derive them. It turns out that it is trivial to use the mechanisms described above to allow the automatic evaluation of terms upon instantiation of their arguments.

Assume that X,8 is bound to (ADD Y 4),7, and that Y,7 is to be bound to the constant 3. Then X,8 is on the list in LINKARRAY(Y,7). When ISSATISFIED inspects that list, after BIND binds Y,7 to 3, it can discover that the value of X,8 is (ADD 3 4). If a procedure for evaluating a ground term with function symbol ADD is available to ISSATISFIED, it could be activated, with arguments 3 and 4, and cause X,8 to be rebound to 7.

Now if X,8 occurred in another term, for example, (GT X 5),8, then the recursive call of BIND used to rebind X,8 to 7, would cause the GT evaluation function to be applied just as the ADD evaluation function was. Thus, (GT X 5),8 would be changed from (GT (ADD Y_7 4) 5) to (GT 7 5) to TRUE, simply by the binding of Y,7 to 3 and evaluation.

It is therefore quite easy to add automatic evaluation of terms whenever they become instantiated in such a way as to allow evaluation to take place. The RESTARRAY and LINKARRAY mechanisms, together with ISSATISFIED in BIND, provide the necessary mechanisms. The only new idea is the rebinding of a variable.

In order to replace (ADD 3 4) by its value, it was essential that it be bound to some variable. This means that input clauses have to be restructured to be lists of literal "templates" with binding environments to properly instantiate them. If evaluation is to proceed all the way to the literals, then even the literals must be variable expressions bound to the desired expressions.

That is, if (+ (GT (ADD X Y) Z)) is an input literal, it would have been previously represented by the list (+ (GT (ADD X Y) Z)). To allow it to be partially evaluated as X, Y, and Z are instantiated, it must be represented by some variable expression, X,n, where X,n is bound to (+ X),m, X,m is bound to (GT X Y),k, and X,k is bound to (ADD X Y),1. Thus, the value of X,n is (+ (GT (ADD X_1 Y_1) Y_k)), and any subterm can be evaluated and replaced by binding. An input clause becomes a list of variable expressions and a list of bindings as in the general resolvent. The maximum index of an input clause is, of course, no longer 1. The function that decomposes a list expression into this form is straightforward.

This modification allows paramodulation to be implemented using structure sharing.

One ramification of this representation is that there now need be only one term for each function symbol, that is, only one (GT X Y) template, and only one (ADD X Y) template. The input atoms:

```
(GT (ADD X O) Z)
```

and

```
(GT X (ADD Y Y))
```

are actually composed of identical terms with different instantiations. Integers become very natural representations for these unique templates.

Evaluation procedures must be associated with function symbols so that ISSATISFIED can detect that a term has an evaluation procedure. In current implementations, the procedure is given the same name as the function symbol, and ISSATISFIED merely checks to see if there is a function in the system with the appropriate name.

The definition of ISSATISFIED is given below. The first half of the function is as in the original definition. The last half is concerned with evaluating terms. The function EVALUATE is used to clarify the code. What this function does is take a variable and index as arguments and attempt to evaluate it. The term to which the variable expression is bound is retrieved and if an evaluation function is associated with its function symbol, this function is called on the expressions representing the arguments. This evaluation function must retrieve the bindings of any variables in these expressions. If the function returns the special atom UNDEF, it means the arguments were not such that the function was applicable (for example, the evaluation function for ADD returns UNDEF if one of the argument expressions is an unbound variable). The special atom FAIL is returned if one of the arguments is "illegal". The evaluation function for DIVIDES returns FAIL if its second argument has the value 0. Otherwise, the evaluation function is assumed to return a new term and index, representing the new value the evaluated expression is to be replaced by.

```
FUNCTION ISSATISFIED V I;
VARS NEWTERM NEWINDEX;
FOREACH RESTFN IN RESTARRY(V,I);
IF RESTFN(V,I) = FALSE
   THEN FALSE; EXIT;
FOREACH PAIR IN LINKARRAY(V,I);
IF ISSATISFIED(FRONT(PAIR),BACK(PAIR)) = FALSE
   THEN FALSE; CLOSE;
CLOSE;
EVALUATE(V,I) -> NEWTERM -> NEWINDEX;
IF NEWTERM = FAIL
   THEN FALSE; EXIT;
IF NEWTERM /= UNDEF
   THEN
   IF BIND(V,I,NEWTERM,NEWINDEX) = FALSE
      THEN FALSE; EXIT;
   CLOSE;
TRUE;
END;
```

An example of an evaluation function for ADD is:

```
FUNCTION ADD ARG1 INDEX1 ARG2 INDEX2;
LOOPIF ISVAR(ARG1) AND ISBOUND(ARG1,INDEX1)
   THEN -> ARG1 -> INDEX1; CLOSE;
LOOPIF ISVAR(ARG2) AND ISBOUND(ARG2,INDEX2)
   THEN -> ARG2 -> INDEX2; CLOSE;
IF ISNUMBER(ARG1) AND ISNUMBER(ARG2)
   THEN
   DUMMYINDEX;
   ARG1 + ARG2;
   ELSE
   DUMMYINDEX;
   UNDEF;
   CLOSE;
END;
```

DUMMYINDEX is just any index whatsoever. An evaluation function must return two arguments (according to the conventions used in ISSATISFIED) and in either case above, the value of the index does not matter.

Note that in the definition of ISSATISFIED, BIND is called to rebind V,I to its new value. This recursive call of BIND may well set off further evaluations (since BIND calls ISSATISFIED to check the repercussions of a binding). It is in this way that the evaluation of a term propagates through all of the terms involving that term.

Rebinding is somewhat complicated because now UNBIND must not merely reset a modified VALUE cell to UNBOUND, but must restore the previous binding. If V,I is bound to T,J and is to be rebound to T',J', then T',J' should be assigned to the TCOMP and TICOMP components of VALUE(V,I). However, an UNBIND must restore T,J. The most natural way to do this is to push T,J onto the BINDSTACK stack, along with the cell modified. UNBIND then simply pops three things off the stack and assigns the last two (T and J) to the appropriate components of the first (the VALUE cell modified). Adding the rebindings to the

BINDINGS component of the new clause record in the normal way allows ISBOUND to correctly fetch the latest binding (as LISP's ASSOC does). However, LOAD must be modified so as to not overwrite a new binding with an older one encountered later in its recursion.

Automatic evaluation can be made to subsume certain restriction functions since ISSATISFIED recognizes the FAIL result of evaluation. Thus, in order to enforce a certain restriction on the syntax of all terms starting with some function symbol, regardless of what clauses are involved in the derivation of the terms, an evaluation function for the function symbol will do the job. For example, in the Blind Hand Problem discussed in the last Section, an evaluation function on DO which returns FAIL whenever the nested situations contain illegal action sequences will cause the same behaviour as the restrictor functions described.

The main point of this Chapter has been to illustrate that structure sharing allows computationally defined symbols to be mixed arbitrarily with logically defined ones. This mixing of computation and deduction allows the axiom writer greater flexibility in controlling the use of his axioms and greater power in specifying the desired behaviour of certain terms. Furthermore, this mix can be efficiently implemented with structure sharing.

CHAPTER 5 THE SL-RESOLUTION IMPLEMENTATION

5.1 Introduction to SL-resolution

Readers familiar with SL-resolution will find the notation in the following description somewhat different from that used by Kowalski and Kuehner (1971). However, it is felt that this notation helps clarify the representation of clauses.

SL-resolution operates on chains. A chain is composed of cells, which contain literals. Cells will be separated by a slash ('/'). The left-most cell in a chain is called the most recent cell. A chain is thus of the form:

A B C / D E /,

where A, B, C, D, and E are literals.

Before a chain may be resolved with an input clause, a literal from the most recent cell must be selected. This selected literal must be the literal resolved upon in all resolutions of the chain. The selected literal of a cell is denoted by underlining it:

A $\underline{B}$ C / D E /.

SL has three operations on chains. Extension corresponds to resolution with an input clause. Reduction is similar to factoring but also replaces ancestor resolution. Truncation is a bookkeeping device for chairs.

To extend upon a chain, C, with selected literal L, an input clause B is used. B must contain some literal K of sign opposite that of L, such that the atoms of K and L unify via most general unifier σ. The result of extending C (the nearparent) by B (the input, or farParent) is the chain C'σ, where C' is the chain whose most recent cell is composed of all of the literals in B except K, and the remainder of C' is just C. For example, extending A $\underline{B}$ C / D E / with the input clause -B G H, yields:

G H / A $\underline{B}$ C / D E /.

The selected literal, L, in C is called an A-literal (A for ancestor) in chain C'. Note that in the above example, either G or H must be chosen as the selected literal before the chain can be used in extension.

A chain is admissible if and only if no two literals in any cells of the chain have identical atoms. A chain may be extended upon only if it is admissible and the most recent cell is non-empty.

A chain C may be reduced if the atom of some literal L in the most recent cell unifies with the atom of an A-literal (other than the most recent) of opposite sign, or a non-A-literal (other than another most recent one) of the same sign. The result of such a reduction is the chain obtained by deleting L from C and applying the unifying substitution.

The first case defined above is called ancestor reduction and performs the function of ancestor resolution in other linear systems. The other case is the usual factoring. Two examples of reduction are:

A B C / $\underline{D}$ B / reduces to A C / $\underline{D}$ B / (factoring)

A B C / $\underline{D}$ E / F $\underline{-B}$ / reduces to A C / $\underline{D}$ E / F $\underline{-B}$ /
(ancestor reduction)

The final operation in SL is truncation. A chain must be truncated when the most recent cell is empty. This condition can be caused by extension with a unit clause, reduction of all the most recent literals, or truncation. The result is simply the chain obtained by deleting the (empty) first cell and deleting the A-literal from the exposed cell (which is now the most recent). Thus,

$/ \ A \ \underline{B} \ C \ / \ \underline{D} \ E \ /$ truncates to $A \ C \ / \ \underline{D} \ E \ /$,

and

$/ \ \underline{B} \ / \ D \ \underline{E} \ /$ truncates to $D \ /$.

An example of an SL derivation follows. Let (1) through (6) be ground input clauses:

(1) -A B

(2) -B C

(3) -C D

(4) -D A

(5) A C

(6) -B -D

Below is a refutation of this set of clauses in SL format:

$-\underline{B} \ -D \ /$	top chain (6)
$-\underline{A} \ / \ -\underline{B} \ -D \ /$	extension with (1)
$-D \ / \ -\underline{A} \ / \ -\underline{B} \ -D \ /$	extension with (4)
$/ \ -\underline{A} \ / \ -\underline{B} \ -D \ /$	reduction
$-\underline{D} \ /$	truncation
$-\underline{C} \ / \ -\underline{D} \ /$	extension with (3)
$-\underline{B} \ / \ -\underline{C} \ / \ -\underline{D} \ /$	extension with (2)
$-\underline{A} \ / \ -\underline{B} \ / \ -\underline{C} \ / \ -\underline{D} \ /$	extension with (1)
$C \ / \ -\underline{A} \ / \ -\underline{B} \ / \ -\underline{C} \ / \ -\underline{D} \ /$	extension with (5)
$/ \ -\underline{A} \ / \ -\underline{B} \ / \ -\underline{C} \ / \ -\underline{D} \ /$	reduction
$\square$	truncation

Of course, in general, the unifying substitution must be applied to chains produced by extension and reduction. Readers interested in a formal account of SL-resolution should consult Kowalski and Kuehner 1971 .

5.2 The Implementation

Before discussing the details of the structure sharing implementation of SL-resolution, several points should be brought out.

(1) Since SL derivations are truly linear, the right parent is always an input clause. Therefore, its maximum index will be 1. When a clause is extended, all of the variables in the right parent will have index MAXINDEX(LEFTPAR(CL)) + 1, where CL is the nearparent.

(2) Since some literal in the most recent cell in any derived clause must be marked as the selected literal, some component of the clause record must be reserved for this. However, no information must be stored in a resolvent's representation describing which literal from the left parent was resolved upon, since it will always be the selected literal of that parent.

(3) Both reduction and truncation remove only literals from the most recent cell. Thus, every cell in an SL chain is just a subset of the literals in the input clause that gave rise to that cell by extension. To specify the literals in a cell a bit string or logical word can be used to "mask" the original input list of literals. Removing a literal will then be simply turning off the appropriate bit in the mask. Another mask can be used to specify the selected literal.

The representation of an input clause is just as it was in the general case. The representation of an SL derived chain is a record with the following components.

(1) the left parent (a clause record) (LEFTPAR)

(2) the input clause supplying the literals in the current cell (RIGHTPAR)

(3) a logical word with bit n on if and only if the nth literal in the list specified by RIGHTPAR is in the current cell (CELLMASK)

(4) a logical word with bit n on if and only if the nth literal in RIGHTPAR is the selected literal for the current cell (SELMASK)

(5) the index for the literals in the current cell (CELLINDEX)

(6) a list of bindings as in the general case (BINDINGS)

If CL is an SL derived clause record, then the components of it describe the leading cell in the chain represented by CL. In particular, each literal in the the cell is represented by an expression of the form T_n,CELLINDEX(CL), where T_n is the nth literal of RIGHTPAR(CL) and the nth bit of CELLMASK(CL) is on. The literal is the selected literal of the cell if the nth bit of SELMASK(CL) is on. The binding environment is, as expected, the BINDINGS of CL along with the binding environment of LEFTPAR(CL).

The basic idea is that a record, CL, should represent a cell and then point, via LEFTPAR, to the chain from which CL was derived. That chain will contain the additional cells in the chain represented by CL. If a cell is to be changed, for example, a literal deleted by reduction, a new record is built to specify the new cell. The old record cannot be modified since it might be used differently in another derivation. But the new record must point back to the old one, to ensure that the bindings there are available. Thus, the cell specified by the old record should not always be included in the chain represented by the new one.

As in the general case, an SL record simultaneously represents an SL chain and the derivation of that chain. The derivation will contain cell descriptions of cells no longer in the chain. Such cells are implicitly marked by their indices, as will become clear in the description of the SL operations, below.

To represent the extension, CL', of some chain, CL, a record is built which refers to the appropriate input parent with RIGHTPAR, and masks off the appropriate literals in it with CELLMASK. CELLINDEX is set to one plus the maximum index found in the records in the structure CL, and BINDINGS are those produced by unification. The LEFTPAR of CL' is CL, indicating that the rest of the cells in CL' are those of CL, and that the rest of the bindings come from CL. SELMASK is set according to whatever literal is to be selected.

To represent a reduction of CL, a new record, CL', is built which has the same RIGHTPAR and CELLINDEX as CL (to specify the same literals as composing the new version of the most recent cell), and CELLMASK is the same except that the single bit corresponding to the reduced literal is turned off. The BINDINGS are produced by unification, and LEFTPAR points to CL to indicate that the rest of the cells in the chain represented by CL' are found in CL (as well as the bindings). However, now CL'specifies the form of the leading cell, and the fact that the indices of CL and CL' are the same indicates that they are different versions of the "same" cell. The cell specification in the record CL will be ignored when exploring for cells in CL'.

To represent a truncation of CL, a new record, CL', is built which has as RIGHTPAR and CELLINDEX the same components of the first record

CL'', in the chain of CL which has a literal in it other than the one marked by SELMASK. The CELLMASK of CL'is the same as for CL'', except that the selected literal bit in it is turned off. BINDINGS is always NIL in a truncation. LEFTPAR points to CL (rather than LEFTPAR(CL'')) since the bindings in the derivation of CL are still relevant. However any cell in the chain of CL with index higher than that of CL'' is ignored since it has been truncated.

Thus, the cells in the chain represented by any CL' are: the cell specified by CL' and the cells in the chain represented by the first record with CELLINDEX strictly less than that of CL' in the list of records linked via LEFTPAR from CL'.

If the CELLINDEX of CL' is strictly greater than that of LEFTPAR(CL') CL' is an extension of LEFTPAR(CL'). If equal, CL' is a reduction of LEFTPAR(CL'). If less than, CL' is a truncation of LEFTPAR(CL').

While the functions BIND, OCCUR, and UNIFY are exactly the same as in the general case, the functions ISBOUND and GETLIT are slightly different (and more efficient).

ISBOUND no longer has to decrement the index of the variable, since it must always loop down the left branch. It never has to be prepared to jump down the right branch, since that is always an input clause.

GETLIT merely runs down the cells in the chain represented by the record specified, counting the number of bits on in the CELLMASKs until it finds the appropriate cell. The literal is then masked out of the parent and the index is just the CELLINDEX of the record involved.

Below is an example of an SL derivation including extension, reduction, and truncation.

```
C1:  ((+ (Q (A) X)) (+ (P Y)))

C2:  ((- (P (F X))) (+ (R X X)))

C3:  ((+ (Q X (F (B)))) (- (R (A) Y)))

C4:  < NIL,C1,110,010,1,NIL >

C5:  < C4,C2,010,010,2,(< Y,1,(F X),2 >) >

C6:  < C5,C3,100,000,3,(< Y,3,(A),3 > < X,2,(A),3 >) >

C7:  < C6,C3,000,000,3,(< X,1,(F (B)),3 > < X,3,(A),1 >) >

C8:  < C7,C1,100,100,1,NIL >
```

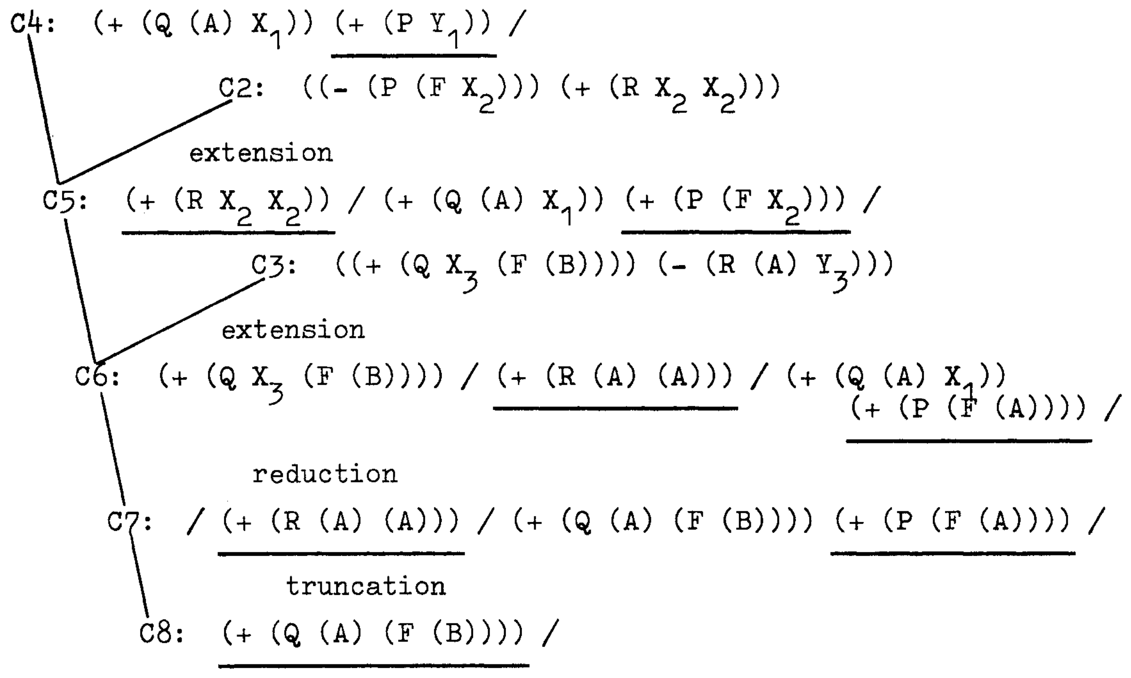

C4: (+ (Q (A) X_1)) $\underline{\text{(+ (P } Y_1\text{))}}$ /
C2: ((- (P (F X_2))) (+ (R X_2 X_2)))
extension
C5: $\underline{\text{(+ (R } X_2\ X_2\text{))}}$ / (+ (Q (A) X_1)) $\underline{\text{(+ (P (F } X_2\text{)))}}$ /
C3: ((+ (Q X_3 (F (B)))) (- (R (A) Y_3)))
extension
C6: (+ (Q X_3 (F (B)))) / $\underline{\text{(+ (R (A) (A)))}}$ / (+ (Q (A) X_1)) $\underline{\text{(+ (P (F (A))))}}$ /
reduction
C7: / $\underline{\text{(+ (R (A) (A)))}}$ / (+ (Q (A) (F (B)))) $\underline{\text{(+ (P (F (A))))}}$ /
truncation
C8: $\underline{\text{(+ (Q (A) (F (B))))}}$ /

Figure 2. (a) At the top are three input clauses and five SL clause records representing the derivation of chain C8. The order of the components in the records is that given at the beginning of this Section. The bit masks have been shortened to only three bits for simplicity in this example. (b) The tree exhibits the SL derivation represented by the records in (a).

The VALUE array is very natural with SL-resolution, since no bindings from the RIGHTPAR are ever loaded. Thus, in a depth-first search, the only way that VALUE is written into is the BIND calls in UNIFY.

The structure sharing implementation of SL is extremely efficient in use of space (in actual implementations, the bit masks are packed into a single word, the CELLINDEX and the specification of the RIGHTPAR are packed into a single word, and the BINDINGS are stored in an array). The program is three times faster than the general structure sharing program, and causes no garbage collections.

Constructing an extension record is as simple as a unification and a few logical operations to construct two masks. Reduction is a unification and turning off one bit in a mask. Truncation is necessary when the CELLMASK is zero and it continues as long as the CELLMASK and the SELMASK of records in the chain are equal. Constructing a truncation record is essentially ANDing the NOT of the SELMASK and the CELLMASK. Sweeping through the literals for reductions or admissibility is the usual TL operation, except that a bit mask is leftshifted each time and the literal is inspected only if the high order bit is on. Thus, the implementation preserves the fast and natural recursive unification algorithm, but allows clauses to be constructed with logical AND, OR, NOT, and SHIFT.

As demonstrated in this chapter, it is often possible to modify general structure sharing to implement specific systems very economically.

CHAPTER 6 A PROGRAMMING LANGUAGE FOR STRUCTURE SHARING

6.1 Introduction

There are several similarities between structure sharing implementations of predicate calculus theorem provers and traditional implementations of programming languages.

Recursion in languages such as LISP and ALGOL uses pushdown stack-like structures to bind variables to different values at different levels in the recursion. When the recursive definition of, say, factorial, is evaluated, the same function body is used for each recursive step. However the argument is bound to a different integer each time, and the binding is looked up in a list, or stack, or dictionary, which has been modified for each recursive call. The alternative is to copy the definition over again each time with the new integer textually substituted, and then use the copy. This is clearly absurd for a programming language, but it is precisely what is done in traditional theorem provers when an axiom is to be applied. Structure sharing takes a programming language approach and merely refers to the appropriate axiom after modifying an association list (the binding environment represented by the tree of BINDINGS) or a dictionary (the VALUE array) to effect variable binding.

In SL-resolution implementations, the clause records resemble activation records, and can be described in such terms: The selected literal mask is the current instruction pointer, relative to the procedure specified by RIGHTPAR. The LEFTPAR component points to the previous level of recursion (in fact, in a depth-first search, the next cell in the chain is the record that should be reactivated if the current computation succeeds, and the record pointed to immediately by LEFTPAR is the record that should be reactivated if the computation fails). BINDINGS specify the bindings of the "local" variables in this procedure call.

However, unlike most programming languages, predicate calculus allows a "function" to be called with free variables in the argument positions, and allows results to contain free "locals" of the function. Thus, space allocated at run-time to hold bindings cannot be garbage collected after a function has been exited, since the results of that function may still contain references to the allocated space. Furthermore, when the interpreter encounters a variable, X, it must know whether it is the X from the current level of recursion or an X from an earlier level. In either case, the X may be free or bound, and if free, it could become bound at the current level.

Thus, a simple pushdown stack allocation scheme is not sufficient. Instead, a system which reflects not just the current "trace" of the computation, but the entire history of it is needed. This is why structure sharing must allocate space for bindings in a tree structure, and keep track of variables by indices rather than simply depth of recursion.

These similarities suggest that there is a programming language natural to structure sharing. The language must allow free variables to occur in arguments and results, and the system must be automatically responsible for looking up the bindings of variables, and keeping track of indices. Access to variables of other levels must be allowed, including levels which have been exited.

The language is, in fact, a simple extension of predicate calculus. A natural interpreter for the language is an SL-resolution theorem prover.

6.2 BAROQUE

An SL theorem prover for interpreting predicate calculus as a programming language has been implemented using structure sharing. Input clauses can only be resolved on their first literal. The literals in a cell are worked on from left to right, and a depth-first search is employed. Evaluation is used to provide basic arithmetic functions, as well as term identity testing with the (predicate calculus function symbol) IDENT. Lists in the language are represented by terms. The list (A B) is the term:

```
          (CONS (A) (CONS (B) (NIL))).
```

Programs are Horn clauses.

The axioms to define LENGTH (which calculates the length of a list in the above representation) are:

```
          LEN1:  ((+ (V (LENGTH (NIL)) O)))
          LEN2:  ((+ (V (LENGTH (CONS X Y)) Z))
                  (- (V (LENGTH Y) U))
                  (- (V (ADD U  1 ) Z))).
```

The predicate symbol V is the only predicate used. (V x y) means "the result of evaluating x is y." The positive (first) literal of a clause specifies the "calling pattern" required to use the procedure. The negative literals represent the sequence of statements to be evaluated as the body of the function. Thus, LEN1 means that the value of (LENGTH (NIL)) is the constant O. LEN2 means that the value of a term of the form (LENGTH (CONS X Y)) is Z, where Z is computed by computing the LENGTH of Y (which is U) and then adding 1 to it.

If the SL theorem prover is given the two axioms above and the top clause:

```
((- (V (LENGTH (CONS (A) (CONS (B) (NIL)))) X))),
```

then it will use the axioms LEN1 and LEN2 exactly as a recursive definition of LENGTH to decompose the list into successive components and calculate that X is 2 (in the process, it will bind X,1 to 2).

Since the axioms used to define programs are Horn clauses and there is only one predicate symbol, a more convenient notation can be adopted. The numbers and the atoms NIL and T can be recognized as constants, and the axioms:

```
((+ (V NIL NIL)))

((+ (V T T)))
```

and

```
((+ (V n n))), where n is a number,
```

are automatically included. The notation used to define LEN1 and LEN2 above becomes:

```
LEN1: (LENGTH NIL) -> 0;

LEN2: (LENGTH (CONS X Y)) -> Z
      WHERE
      (LENGTH Y) -> U;
      (ADD U 1) -> Z;
      END;
```

This language is called BAROQUE. It has several properties not found in traditional programming languages. Among these are: pattern directed invocation and return, backtracking, and the ability to run functions "backwards" (from results to arguments).

If the theorem prover (interpreter) is given as top clause:

```
WHERE (LENGTH (CONS (A) NIL)) -> X; END;
```

the following sequence of operations occurs. LEN2 is used to extend with the top clause (LEN1 is tried but does not unify because NIL will not unify with (CONS X Y)). This produces a chain in which the leading cell contains two literals. The first is a recursive call of LENGTH on NIL, giving the value U,2, and the second is (ADD U,2 1) yielding the value X,1. The first literal is extended upon with LEN1, binding U,2 to 0. After a truncation (because LEN1 is a unit clause) the second literal is extended upon (with the ADD evaluation function). Since U,2 is now 0, this binds X,1 to 1. Now both literals of this cell have been deleted, so truncation yields the empty clause and the computation is done. The answer, X,1, is 1. No search is performed.

Since unification is the only thing really happening, pattern directed invocation and returning is clearly present. Regardless of how many definitions of LENGTH there may be, only those which unify with the calling pattern are ever used. Thus, if LEN3 is added:

```
LEN3:  (LENGTH (TRIP X Y Z)) -> U
       WHERE
       (LENGTH Y) -> Y1;
       (LENGTH Z) -> Z1;
       (ADD Y1 Z1) -> U;
       END;
```

then a call of the form:

```
(LENGTH (TRIP ...)) -> X;
```

will only be handled by LEN3, since LEN1 and LEN2 do not unify.

As for the ability to run functions backwards, consider a call of the form:

```
(LENGTH X) -> 2;
```

(and assume LEN3 has not been added, just for simplicity).

On the first extension with LEN2, the value of X,1 will be:

(CONS X,2 Y,2).

A second extension with LEN2 causes X,1 to become:

(CONS X,2 (CONS X,3 Y,3)).

Finally, LEN1 can be used to produce:

(CONS X,2 (CONS X,3 NIL)).

LEN1 could not be used earlier because (on the first extension) it would attempt to unify 0 and 2, or (on the second extension) it would attempt to unify 1 and 2. That is, the expected result must unify with the result provided by any function. If it does not, the extension fails and another is tried. Note that what the above process has done is to generate the general list of LENGTH 2.

In general, BAROQUE will explore all possiblities in trying to find a successful path through the search space (to the empty clause). Since a depth-first search is used, infinite branches are easy to get lost on. However, since BAROQUE is actually just an SL-resolution theorem prover, there is a clear distiction between the logical features of the language and the search strategic ones.

6.3 A LISP Subset in BAROQUE

BAROQUE is very much like assembler-code. Consider the BAROQUE function for calculating $(X^2 + 1)/2$:

```
POLY: (POLY X) -> U
      WHERE
      (MULT X X) -> V;
      (ADD V 1) -> W;
      (DIV W 2) -> U;
      END;
```

A call of the form:

```
          (POLY 4) -> X;
```

binds X (at the current level) to 8.5.

However, a call of the form:

```
          (POLY (POLY 4)) -> X;
```

fails, because the inner (POLY 4) is not evaluated. To get the desired effect in BAROQUE, one must write:

```
          (POLY 4) -> X1:
          (POLY X1) -> X;
```

However, it is possible to write an interpreter for another language in BAROQUE, and in this language use nested expressions. As an example, a list processing language based on pure LISP was written. The BAROQUE functions are listed below:

```
          CONS: (CONS X Y) -> (CONS U V)
                WHERE
                X -> U;
                Y -> V;
                END;

          CAR: (CAR X) -> U
               WHERE
               X -> (CONS U V);
               END;

          CDR: (CDR X) -> V
               WHERE
               X -> (CONS U V);
               END;

          COND1: (COND X Y Z) -> U
                 WHERE
                 X -> NIL;
                 Z -> U;
                 END;

          COND2: (COND X Y Z) -> U
                 WHERE
                 X -> V;
                 (IDENT V NIL) -> NIL;
                 Y -> U;
                 END;
```

```
EQUAL: (EQUAL X Y) -> U
       WHERE
       X -> V;
       Y -> W;
       (IDENT V W) -> U;
       END;

ADD1: (ADD1 X) -> U
      WHERE
      X -> V;
      (ADD V 1) -> U;
      END;
```

One can then define LENGTH and MEMBER as follows:

```
LENGTH: (LENGTH X) -> U
        WHERE
        (COND X
              (ADD1 (LENGTH (CDR X)))
              0) -> U;
        END;

MEMBER: (MEMBER X Y) -> U
        WHERE
        (COND Y
              (COND (EQUAL X (CAR Y))
                    T
                    (MEMBER X (CDR Y)))
              NIL) -> U;
        END;
```

Arguments are evaluated as required and functions can be nested to any depth. It should be noted that the unusual features of BAROQUE are inherited by this language. For example,

```
(LENGTH X) -> 2;
```

will still generate the general list of LENGTH 2, even with the above definition of LENGTH.

The use of BAROQUE as a programming language, interpreted by an SL-resolution theorem prover and running under a structure sharing implementation which closely models conventional programming language implementations, demonstrates the essential equivalence of computation and deduction in practical terms.

END OF PART I

INTRODUCTION TO PART II

This paper describes an automatic theorem prover which is capable of producing inductive proofs of a large number of interesting theorems about functions written in a subset of pure LISP. The program was designed to prove theorems in the way a good programmer might intuit them. It has several features which make it distinct from other systems concerned with proof of program properties: It is fully automatic, requiring no information from the user except the LISP definitions of the functions involved and the theorem to be proved. It automatically uses structural induction when necessary, and automatically generates its own induction formulas. It will occasionally generalize the theorem to be proved, and in so doing, often "discovers" interesting lemmas. Finally, it is capable of writing new, recursive LISP functions to help properly generalize a theorem.

The primitives in the LISP subset are: NIL, CAR, CDR, CONS, EQUAL, and COND. The user can define and use any number of recursive functions. Theorems take the form of universally quantified boolean valued LISP expressions, and the theorem prover tries to establish that the expression will evaluate to T whenever the quantified variables are replaced by arbitrary lists.

Not only may defined functions call other defined functions, but the statement of the theorem to be proved may contain as many new functions as necessary to capture the concepts needed. The system has no built-in information about non-primitive functions except their definitions. It relies entirely upon these definitions to discover and prove properties needed. No lemmas are used.

The functions AND, OR, NOT, and IMPLIES may be defined in terms of COND, and provide the necessary logical facilities. The system only knows the definitions of these functions; its logical behaviour is based on its knowledge of COND. The fact that no predicates or relations (other than EQUAL) are built-in has two implications: The system must be (and is) powerful enough to derive and prove facts about many non-primitives in the course of proving a single theorem, since it must derive the necessary properties of the predicates and auxiliary functions involved. Secondly, the system is very extendable. In particular, the language in which theorems are stated can be expanded at will by the user (with recursive functions), rather than being limited to a collection of built-in predicates and relations.

Functions about which theorems have been automatically proved include most of the elementary LISP functions: APPEND, ASSOC, FLATTEN, LENGTH, MEMBER, NUMBERP, OCCUR, PAIRLIST, PLUS, REVERSE, SUBST, TIMES, UNION, and many others. The functions and their definitions are listed in Appendix A. Theorems automatically proved are listed in Appendix B. Except for the definitions in Appendix A, no other information about the functions involved is known to the system.

As an example of the complexity of some of the theorems, consider the program's proof that a list sorting function is correct. There are two theorems involved, one which states that the output of the sorting function is ordered, and another which states that the sorted list has exactly the same elements in it as the unsorted one. There are five function definitions involved: a function which determines if one list is less than or equal to another (in length), LTE; a function which returns T if and only if its argument is an ordered list, ORDERED; a function which adds a new element to an ordered list, so that the result is ordered, ADDTOLIST; a function which uses ADDTOLIST to sort a list, SORT; and a function which counts the number of occurrences of an element in a list, COUNT.

The two theorems which establish the correctness of the SORT function are:

(ORDERED (SORT A)),

and

(EQUAL (COUNT A B) (COUNT A (SORT B))),

where A and B represent universally quantified variables (skolem constants). The theorem prover uses induction to prove that both of these expressions evaluate to T, regardless of the values of A and B.

The proofs are very intuitive. For example, in the first theorem above, the program does an induction on A and after some rewriting and generalization, reduces the theorem to:

(IMPLIES (ORDERED C) (ORDERED (ADDTOLIST D C))),

which is one of the basic properties of ADDTOLIST (although that fact is certainly not known to the system). It then proves this by induction on C, using a very interesting induction formula. Again, after much simplification, it produces the following theorem which must be proved:

(OR (LTE D E) (LTE E D)).

This of course is an elementary fact about LTE, but it must be proved since LTE is a user-defined function. To prove this, the program inducts simultaneously on D and E. As an indication of the performance level of the program, the entire process of showing that the output of SORT is ORDERED requires 41 seconds.

The basic idea behind the automatic generation of the induction formula is that there is a duality between recursion and induction. When the program must resort to induction, it passes the theorem to be proved to a modified LISP interpreter which attempts to evaluate the expression. The evaluation halts when a function tries to recurse into the structure of an argument whose structure is not well enough defined to allow the computation of the requisite substructure. For example, the evaluation of (APPEND A B) halts because APPEND attempts to CDR into A in the recursive call, and the structure of A is such that the CDR of A is not available explicitly. The interpreter communicates this information to the induction routine, which attempts to generate an induction formula which will have as its hypothesis the statement that the theorem holds for the requisite substructures, and as its conclusion, that it holds for the structure being decomposed in the recursion. Care is taken that the necessary bases are included so that this is indeed a valid induction formula.

Thus, the induction formula generated depends upon the type of recursion used by the functions concerned. The theorem prover is therefore very flexible in its selection of the induction argument to use. For example, it will induct upon n variables simultaneously, or upon tree-structures, as the occasion demands.

Since the program was designed to be able to prove simple theorems in the same way a good programmer might, its methods are easily understood by readers not familiar with other work in the theory of computation.

Largely because of its straightforward methods, the program is surprisingly fast by theorem proving standards. The "typical" theorem requires about 10 seconds to prove. Harder ones, such as the COUNT theorem mentioned above, require more time (150 seconds for the COUNT theorem).

The organization of this presentation is as follows: Chapter 1 lays the formal foundations. Here we present a set of axioms defining a first order theory with a syntax modeled on LISP. The primary difference between the language used here and the LISP subset with the same set of primitives is the conditional statement. The COND used in this paper has three arguments; the first is the condition tested, and the other two are the "branches" of the tree. Thus, the statement:

```
          (COND p q (COND r s u)),
```

in the language used here, represents the LISP statement:

```
          (COND (p q) (r s) (T u)).
```

Section 1.1 may be skipped by those with a knowledge of LISP who are not interested in the formal details.

Chapter 2 discusses the relation between evaluation and induction which is used to produce induction formulas. This chapter contains two examples, in English, of proofs produced by the program.

Chapter 3 is a detailed description of the program. It is broken into sections according to subroutines in the system. Section 3.1 presents a good overview of the system. Section 3.2 describes the evaluation machinery. Section 3.7 discusses the generalization heuristic and the automatic programming feature. Section 3.8 presents the induction mechanism.

Chapter 4 exhibits four proofs produced by the program. Among the proofs shown is that of the theorem:

(ORDERED (SORT A)),

described earlier.

Chapter 5 discusses two desirable extensions of the current program. One of these is the automation of termination proofs, and the other is extending the LISP subset to include PROG, SETQ, GO, and RETURN.

Chapter 6 concludes the presentation with a discussion of the kinds of information built into the program, the program's ability to generate natural lemmas, and the design philosophy of the system.

Appendices A and B contain the definitions of the LISP functions about which theorems have been proved, and some of the theorems proved by the program. Appendix C presents the theory of lists (in which the theorems are proved) as an extension of number theory (to establish its consistency). Appendix D is a survey of other work in the field.

Readers interested in obtaining a quick overview of the system are advised to read Chapters 2 and 6, Section 3.1, and Appendix B.

The notational conventions used in this document should be explained before proceeding.

When presenting terms in theorems or formulas, both formally and informally, the upper case letters X, Y, Z, U, and V will be used to denote variables. Occasionally integers will be concatenated with these letters to expand the class of variables. Thus, X and Y2 are variables which range over the set of individuals.

Function names will always be words in upper case. The letters A, B, C, D, and E will denote universally quantified variables in the theorem to be proved (skolem constants). Traditional LISP notation will be used to represent function application. Thus,

    (FOO NIL (BAR A))

represents the application of the function FOO to two arguments: NIL and BAR applied to A.

The infix equality predicate, =, will be used. The infix symbols &, v, ¬, ->, and <->, will denote the usual logical connectives. A negative equality will be abbreviated to ≠. An example formula is:

    ¬ (MEMBERP A B) -> (B = NIL v A ≠ (CAR B)),

provided MEMBERP is a defined predicate symbol.

In order for us to talk about terms and formulas we will need 'syntactic variables'. These are not part of the theory but are additions to English, and take as values expressions in the theory. Lower case letters (possibly subscripted) will be used as such variables. Syntactic variables will generally be used to describe classes of expressions. Thus, we will say that

    (CONS (CAR A) NIL),

and

    (CONS (CONS A B) NIL),

are both of the form (CONS x NIL).

The syntactic variables f, g, and h (possibly subscripted) will range only over function names.

If we say that p(x,y) is some expression q in which x and y appear (possibly no times), then p(A,B) is q with all occurrences of x replaced by A, and all occurrences of y replaced by B. Thus, if p(x) is (CONS x NIL), then

    (CONS p(A) p(B))

is

    (CONS (CONS A NIL) (CONS B NIL)).

Finally, the procedures which make up the theorem prover will be called routines. Whenever the name of a routine or of an identifier in a routine is used, it will be in lower case and underlined. As part of a theorem prover, these routines often take expressions in the theory as arguments, and return new expressions as results. The notation eval(x) will be used to denote the result of applying the routine eval to the expression represented by x.

If eval is some routine with input (f x_1 ... x_n), the phrase "the input expression with its arguments eval'ed" refers to the new expression:

    (f eval(x_1) ... eval(x_n)).

In particular, it does not refer to the actual list structure representing the input expression after its argument positions have been destructively altered.

CHAPTER 1 FOUNDATIONS

1.1 The Theory of Lists

If we wish to formally prove theorems, we must define the theory in which our "theorems" are theorems. The theory with which we are dealing is the first order theory of lists. This theory is very similar to LISP. Readers familiar with LISP and uninterested in formal details may skip this section.

The non-logical symbols of our theory are the constant NIL, the unary function symbols CAR and CDR, the binary function symbols CONS and EQUAL, and the ternary function symbol COND. We will call these symbols the 'primitives'. Except for the fact that COND has three arguments here, these functions behave in the same spirit as do their counterparts in LISP.

The non-logical axioms of the theory are given below. All variables are universally quantified.

NIL ≠ (CONS X Y).

(CONS X Y) = (CONS U V) <-> X = U & Y = V.

(CAR NIL) = NIL.

(CDR NIL) = NIL.

(CAR (CONS X Y)) = X.

(CDR (CONS X Y)) = Y.

(EQUAL X Y) = (CONS NIL NIL) v (EQUAL X Y) = NIL.

(EQUAL X Y) = (CONS NIL NIL) <-> X = Y.

(COND (CONS X Y) U V) = U.

(COND NIL U V) = V.

In addition, we include the induction axiom for each formula p(x):

$$p(NIL) \;\&\; \forall X,Y(p(X) \;\&\; p(Y) \rightarrow p((CONS\ X\ Y))) \rightarrow \forall X(p(X)).$$

For notational simplicity we will use the symbol T as an abbreviation for the term (CONS NIL NIL). Thus, the axioms for EQUAL can be abbreviated to:

$$(EQUAL\ X\ Y) = T \vee (EQUAL\ X\ Y) = NIL.$$

$$(EQUAL\ X\ Y) = T \;\leftrightarrow\; X = Y.$$

We will refer to T and NIL as 'truthvalues', not to be confused with logical truth and falsity. An expression will be said to be 'boolean' if it is always equal to a truthvalue. For example, any expression with function symbol EQUAL is boolean, by the first equality axiom above.

We will say that a term 'is a CONS' if it is of the form (CONS x y). Analogous phrases will be used for the other primitive function symbols. A term of the form (COND x u v) will be called a 'conditional expression'. The first argument, x, is called the 'test'. The second, u, is called the 'true-branch', and the third, v, is called the 'false-branch'. By the two COND axioms, (COND x u v) is always equal to u or v, depending upon x. The name 'true-branch' is perhaps a misnomer, since (COND x u v) is equal to u if x is any CONS whatsoever, not just T.

We say that an expression is an 'explicit list' if it is NIL or a CONS. One immediate consequence of the induction axiom is the theorem:

$$X = NIL \;\vee\; X = (CONS\ (CAR\ X)\ (CDR\ X)).$$

That is, in this theory, everything is equal to some explicit list.

We will say that a term is a 'specific list' if it is an explicit list and every subterm of it is an explicit list. Thus, a term is a specific list if it is composed only of NIL and CONS expressions.

It is desirable to have a representation of the natural numbers among our terms. We will therefore agree that 0 shall be represented by NIL, and that if some natural number, n, is represented by x, then the successor of n is represented by (CONS NIL x). Furthermore, we will agree to abbreviate the terms representing the natural numbers by the numbers they represent. Thus, 3 is our abbreviation for the (specific) list:

```
        (CONS NIL (CONS NIL (CONS NIL NIL))).
```

We will call such terms 'numbers'.

List theory can easily be seen to be consistent. (Its consistency is demonstrated in Appendix C, where it is exhibited as an extension of number theory.)

We now wish to extend the theory with the addition of defined functions. Such functions will be called 'non-primitives'. Consider the defining axiom for the function AND:

```
        (AND X Y) = (COND X (COND Y T NIL) NIL).
```

Since the existence and uniqueness conditions for AND are met, namely:

```
        ∃Z(COND X (COND Y T NIL) NIL) = Z,
```

and

```
        (COND X (COND Y T NIL) NIL) = Z1  &
          (COND X (COND Y T NIL) NIL) = Z2   ->  Z1 = Z2,
```

we can add the defining axiom to the theory and preserve consistency. In fact, the resulting extension of the theory is conservative. (An extension is conservative if no new theorem is provable in the extension except ones involving the new function symbol.)

It is possible to prove in the extended theory the theorems:

```
(AND X Y) = T  <->  (X ≠ NIL  &  Y ≠ NIL),
```

and

```
(AND X Y) = NIL  <->  (X = NIL v Y = NIL).
```

Thus, if we have a formula of the form:

```
p = T & q = T,
```

where p and q are boolean, it can be replaced by:

```
(AND p q) = T,
```

justifying the choice of the name "AND". The definition of AND is such that any term with function symbol AND is boolean, regardless of the values of the arguments.

We can introduce the definitions of OR, NOT, and IMPLIES in a similar fashion, and prove the theorems justifying their names. The reader is referred to Appendix A for their definitions.

However, we also want to add such definitions as:

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

That is, we wish to be able to extend the theory with the introduction of recursive functions. The problem, of course, is preserving consistency. If we allow the addition of arbitrary axioms purporting to define "functions", we immediately open the door to inconsistency. Consider the "function" defined by:

```
(RUSSELL X) = (COND (RUSSELL X) NIL T).
```

If we add this, we can prove:

```
(RUSSELL X) = NIL  <->  (RUSSELL X) = T,
```

which implies NIL = T. Since our stated aim was to prove valid theorems, and to do so meaningfully one must have a consistent theory, such definitions cannot be allowed.

However, as demonstrated in Appendix C, we can be sure that the resulting extension is consistent (in fact, conservative) if the defining axiom for the new function symbol is one of several forms. One such form, describing many common function definitions, is:

```
(f X Y) = (COND X
                (h (f (CAR X) Y) (f (CDR X) Y) X Y)
                (g Y)),
```

where h and g are either primitive functions, or have already been introduced in this way. This is just one version of the primitive recursive schema for list theory.

We are not limited to primitive recursive functions however. As made clear in Appendix C, we can add any total recursive function and be guaranteed that the resulting extension is conservative. In general, we merely want to keep the extension consistent. Totality guarantees it. This is discussed further in Section 5.1.

Most of the functions in this paper are in fact primitive recursive. For example, the definition of APPEND, exhibited above, is in this schema.

Intuitively it is clear that functions in the schema above are total functions, provided h and g are total functions. Assume that whenever their arguments are well-defined, h and g are well-defined. Then f is well-defined by the following inductive argument: (f NIL Y) is well-defined, for any such Y, since it is equal to (g Y). Inductively assuming that (f A Y) and (f B Y) are well-defined, we must show that (f (CONS A B) Y) is well-defined. But this is just:

    (h (f A Y) (f B Y) (CONS A B) Y),

by the axioms of the theory. Since our induction hypothesis tells us the arguments to h, above, are well-defined, and we know that h is well-defined when its arguments are, we conclude that (f (CONS A B) Y) is well defined. Thus, f is well-defined.

We allow a new predicate, p, to be introduced only by an axiom of the form:

    (p X1 ... Xn) <-> (f X1 ... Xn) = T,

where f is a previously introduced, boolean valued, total function. This allows the introduction of a great many predicates, namely, any predicate whose characteristic function is a total, recursive function.

It should be noted that EQUAL can be recursively defined in terms of the other primitives and recursion (see the definition of EQUALP in Appendix A). Hence, EQUAL is primitive only in the sense that it is built into the definition of the basic theory, and into the theorem prover. This is reasonable since it is the characteristic function of the only predicate in the theory, equality. The senses in which EQUAL is built-in are discussed in Section 6.1.

Henceforth we will consider the theory being discussed to be that of lists, extended by the forty or so function definitions listed in Appendix A. These functions include the standard arithmetic and list processing functions.

There are a large number of interesting theorems in this theory. For example:

```
(APPEND X (APPEND Y Z)) = (APPEND (APPEND X Y) Z).
(REVERSE (REVERSE X)) = X.
((MEMBERP X Y) v (MEMBERP X Z)) -> (MEMBERP X (UNION Y Z)).
(ORDERED (SORT X)) = T.
```

Except for the fact that the functions introduced are very common, there is nothing special about the extension of list theory with which we are dealing. The theorem prover embodies the non-logical axioms of the theory and several theorems derivable from these axioms. However, no information about non-primitive functions is available except the defining axioms.

To exhibit the definition of a new function, f, we will write:

(f (LAMBDA (x_1 ... x_n) defn)),

where the x_i represent variables. This will be equivalent to adding the defining axiom:

(f x_1 ... x_n) = defn.

It is assumed that no other axiom defining f has been introduced. The notation above was adopted because it is consistent with LISP. (The routine define takes as its argument a list expression in the form above, and stores the definition of f on the property list of the word f. This effectively extends the theory with which the theorem prover is dealing.)

The definition of APPEND is thus:

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

The theorem prover is capable of inspecting and applying these axioms when necessary. The particular extension of the theory in which the theorem prover operates is determined by the user of the program, who supplies the defining axioms for the non-primitive functions.

The only theorems we will prove with the program are those of the form:

$\forall$X1, ... Xn(p),

where p has no quantifiers and contains only the variables X1, ... Xn. That is, we will not automatically prove theorems which cannot be expressed as universally quantified statements. This excludes a large number of interesting theorems.

If the theorem to be proved is of the form given above, we can rewrite it as:

q = T,

where q is obtained from p by replacing the variables X1, ... Xn by skolem constants, = by EQUAL, the predicates by their characteristic functions, and any logical connectives by the defined functions AND, OR, NOT, and IMPLIES. This is always possible since the characteristic functions have always been introduced, and the functions replacing the connectives have the desired properties.

The input to the theorem prover is then simply q. The program tries to establish that the expression is equivalent to T, and if it does so, the theorem is proved.

For example, to prove the theorem:

((MEMBERP X Y) v (MEMBERP X Z)) -> (MEMBERP X (UNION Y Z)), we present the program with:

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

where MEMBER is the characteristic function for the predicate MEMBERP.

This approach to formalizing a theory is the traditional first order predicate calculus one. It is clearly explained by Shoenfield (1967). There are many alternative approaches. Goodstein (1957) describes a very elegant technique based upon recursive functions. The approach taken here was chosen because it was consistent with the developmental history of the project, and because it is the most widely known. A theory is necessary at all for the following three reasons: It allows the methods used by the program to be explained, justified, and understood. It makes it clear that the techniques used by the program, especially the induction mechanism, are just as applicable to many other theories (in particular, number theory). Finally, it convinces the reader that the program is indeed a theorem prover for a first order mathematical theory, rather than simply a program which manipulates expressions without regard for logical validity.

1.2 LISP and the Theory of Lists

In Appendix C a number theoretic version of list theory is exhibited. This is done to establish the consistency of the theory and extensions of it by total, recursive functions.

However, from the non-logical axioms it is obvious that the primitives in the theory are a first order formalization of a subset of pure LISP. In particular, the theory models the subset composed of the single atom NIL, the functions CAR, CDR, CONS, and EQUAL, and the conditional statement COND.

The fact that only one atom is available does not weaken the language. We still have a countably infinite number of distict objects. As we have seen, the natural numbers are available. Suitable conventions can make other "atoms" available. Other programming languages are quite convincing in their illusion that structures other than binary digits are available.

Terms in the theory represent well-formed LISP S-expressions on the above alphabet in the obvious way. For simplicity, the COND function has a fixed number of arguments. Thus, the LISP statement:

```
(COND (p q) (r s) (T u))
```

is represented by:

```
(COND p q (COND r s u)),
```

in the theory.

The user is allowed to define new functions. As in many languages, the mechanisms for defining a new function are not provided in the language itself (unlike real LISP). When we say that total functions may be introduced consistently, we are saying that the theory consistently models those LISP functions which always terminate without error when evaluated on arbitrary arguments. The user must satisfy himself as to the consistency of any extension produced by adding a partial function.

If two S-expressions evaluate to identical S-expressions under the LISP interpreter for this subset, then the terms representing those two expressions in the theory are equal. For example,

```
(APPEND (CONS (CONS NIL NIL) NIL) (CONS NIL NIL))
```

and

```
(CONS (CONS NIL NIL) (CONS NIL NIL))
```

both evaluate to the S-expression:

```
((NIL . NIL) . (NIL . NIL)).
```

We therefore expect that

```
(APPEND (CONS (CONS NIL NIL) NIL) (CONS NIL NIL)) =
(CONS (CONS NIL NIL) (CONS NIL NIL))
```

is a theorem. This is indeed the case.

Furthermore, if two terms in a consistent extension of the theory are equivalent, then uniformly replacing the universally quantified variables by specific lists and evaluating the corresponding S-expressions gives identical results whenever both evaluations terminate. The only exception to this rule is that, in the theory, CAR and CDR are defined at NIL, while in LISP they are not. This addition to the theory is made to guarantee that the primitives are total functions.

Since we can prove the associativity of APPEND in the theory:

```
(APPEND X (APPEND Y Z)) = (APPEND (APPEND X Y) Z),
```

we expect that

```
(APPEND X (APPEND Y Z))
```

and

```
(APPEND (APPEND X Y) Z)
```

always evaluate to identical S-expressions under a LISP interpreter, regardless of the values of X, Y, and Z. This of course is also the case.

The observations of this section are far from profound. The point is merely that in addition to proving theorems in a first order theory, we are proving theorems about a non-trivial collection of LISP programs.

CHAPTER 2 PROVING THEOREMS IN THE THEORY

2.1 Evaluation, Recursion, and Induction

Central to this thesis is the concept of evaluation. This concept usually resides in the semantics of a theory. One can talk about computing the value of a function applied to some arguments by using the recursive definition of the function. There is, of course, a perfectly parallel version of evaluation in the syntax. There, one talks about deriving the expression representing the value of an expression representing a function applied to its arguments.

Not surprisingly, the process by which this is done is exactly the same: Use the non-logical axioms to derive the values of primitive expressions, and evaluate a non-primitive function application by replacing the variables in the definition of the function by the values of the corresponding arguments, and then evaluating the result.

Evaluation will be discussed at length later, but here it is important to get an intuitive grasp of what is meant by it. As indicated above, the best model for it is simply an interpreter for terms in the theory. We thus evaluate (CAR (CONS A B)) to get A. Evaluating the expression:

```
(APPEND (CONS A NIL) (CDR (CONS B C)))
```

yields (CONS A C). We obtain this result as follows.

We first evaluate the two arguments to the APPEND. We consider the value of the first to be (CONS A NIL), but the second argument evaluates to simply C. We then substitute these two values for the variables in the definition of APPEND and obtain:

```
(COND (CONS A NIL)
      (CONS (CAR (CONS A NIL)) (APPEND (CDR (CONS A NIL)) C))
      C).
```

Evaluating this, we use the first COND axiom to observe that it is equivalent to the value of its second argument:

```
(CONS (CAR (CONS A NIL)) (APPEND (CDR (CONS A NIL)) C)).
```

We therefore evaluate both arguments to this CONS. The first is simply A. The second is another non-primitive expression, so its arguments are evaluated, and then substituted into the definition of APPEND. This gives us:

```
(COND NIL
      (CONS (CAR NIL) (APPEND (CDR NIL) C))
      C).
```

This time the second COND axiom is used to reduce this to C. Thus, the value of the second argument to the CONS above is simply C. At this point we stop (rather than construct a list cell with A in the first half and C in the second) since we are staying in the syntax of the theory. Therefore, the result of evaluating the original APPEND term is (CONS A C).

It should be clear that we are simply interpreting terms in the theory exactly as we would the LISP expressions they represent. Since we are always replacing terms by equivalent ones, we observe that since

```
(APPEND (CONS A NIL) (CDR (CONS B C)))
```

evaluates to (CONS A C), the two expressions are equal in the theory. Thus, if some expression, p, evaluates to T, then p = T is a theorem.

Evaluation is sufficient to prove some trivial theorems. For example:

```
(EQUAL (APPEND NIL A) A)
```

evaluates to T, so we have proved:

(APPEND NIL X) = X.

However, induction is necessary for most interesting theorems.

It is intuitively clear that evaluation and induction are complementary. The paradigm for evaluating a simple recursive function, f, is: evaluate (f (CONS x y)) in terms of (f x) and (f y), and handle the 'NIL-case', (f NIL), separately. But the paradigm for a simple inductive proof that (f X) is T for all X is: show that (f NIL) is T, and then assuming that (f A) and (f B) are T, show that (f (CONS A B)) is T.

In particular, evaluation of a recursive function 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, (f (CONS A B)) = T, to a statement involving the induction hypotheses, (f A) = T and (f B) = T, provided the (CONS A B) is one of the structures that f decomposes in its recursion.

Suppose that we wish to prove by induction that (f X) is T for all X. To show that (f NIL) is T, the obvious thing to do is to evaluate it and see. Provided we establish the NIL-case, we then assume (f A) and (f B) are T, and try to show that (f (CONS A B)) is T. Evaluating the conclusion should give us some expression, q((f A),(f B)). But the inductive hypotheses tell us that (f A) and (f B) are T, so we must then just show that q(T,T) is T. This process is illustrated by the examples in the next two sections.

Of course, if f 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 upon the structures on which f recurses, that is, upon the structures that are being recursively decomposed in the evaluation of f. This ensures that when the induction conclusion, (f (CONS A B)), is evaluated, f will be able to recurse at least one step, and the problem will be transformed to one involving the induction hypotheses.

2.2 An Example of Evaluation and Induction

To illustrate how evaluation and induction can be used together to produce a proof, we will work through the program's proof that APPEND is associative.

The statement of the theorem is:

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

where the definition of APPEND is:

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

Evaluating (1) leads nowhere, however, from trying to evaluate it we learn which terms are being recursively decomposed: A is being decomposed in the calls (APPEND A (APPEND B C)) and (APPEND A B); B is being decomposed in the call (APPEND B C); (APPEND A B) is being decomposed in the call (APPEND (APPEND A B) C). Since we cannot induct upon (APPEND A B), we do not consider it as a possible induction candidate. This leaves induction on either A or B. We choose A because it is recursed upon the most often.

First, we must prove the NIL-case, which is just (1) with A replaced by NIL:

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

But this just evaluates to T, because both arguments to the EQUAL evaluate to (APPEND B C), and EQUAL evaluates to T if its arguments have identical values.

So we must now prove the induction step. We will assume:

```
(2)       (EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C)),
```

as our hypothesis, and try to prove:

```
(3)       (EQUAL (APPEND (CONS A1 A) (APPEND B C))
                 (APPEND (APPEND (CONS A1 A) B) C)).
```

Logically, we could assume the induction hypothesis for A1 as well as A. But since the recursion in APPEND is on the CDR of its first argument, we can predict that such a hypothesis will not be needed. The evaluated conclusion should involve APPEND applied only to the CDR of (CONS A1 A), not to the CAR as well. Thus, we use the simple hypothesis, (2).

If we evaluate the arguments to the EQUAL in the conclusion, (3), they become:

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

and

```
          (CONS A1 (APPEND (APPEND A B) C)).
```

(For the first argument, this evaluation should be obvious. For the second, note that the inner APPEND decomposes the (CONS A1 A) in its first argument and produces, as its result, the value:

```
          (CONS A1 (APPEND A B)),
```

which is the first argument to the outer APPEND. Since it is evaluated after its arguments have been, it now decomposes the CONS supplied in its first argument and produces the result above.)

Once the two arguments to the EQUAL are evaluated, the (built-in) definition of EQUAL is applied. Since two CONSes are EQUAL if and only if their CARs and CDRs are EQUAL, according to the non-logical axioms, the system compares the corresponding components. The CARs are identical (both are A1), but the CDRs are not. Thus, the equality of the two rests on the truth of the statement that their CDRs are equal:

(4) (EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C)).

But (4) is just the induction hypothesis, (2), which we are inductively assuming to be T. So we have proved that (3), the induction conclusion, is T, using the induction hypothesis and evaluation. Hence, the associativity of APPEND has been established.

By inducting upon the structure that was being recursed upon we ensured that the induction conclusion could be evaluated at least one step. This was supposed to yield a simple expression involving the induction hypothesis. In this particular theorem, it yielded the induction hypothesis itself.

2.3 An Example of Additional Techniques

The previous example was very simple for two reasons: The NIL-case was trivial (it evaluated to T), and the induction hypothesis was easy to use (it was identical to the evaluated conclusion). The next example illustrates several techniques that are useful in more complicated situations. Once these techniques have been illustrated, the theorem prover can be described.

The theorem to be proved is the following interesting relationship between APPEND and REVERSE:

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

where the definition of REVERSE is:

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

Again, evaluation of (1) leads nowhere, but it indicates that B is being recursed upon (twice). We therefore induct on B.

The NIL-case evaluates to:

```
(2)       (EQUAL (APPEND (REVERSE A) NIL) (REVERSE A)).
```

We must now prove this before proceeding to the inductive step. However, the term (REVERSE A) occurs on both sides of the EQUAL. This suggests that we might try generalizing the theorem and proving:

```
(3)       (EQUAL (APPEND C NIL) C).
```

Formally, this corresponds to proving:

$$\forall X(\text{APPEND (REVERSE X) NIL}) = (\text{REVERSE X}),$$

by proving the more general theorem:

$$\forall Y(\text{APPEND Y NIL}) = Y.$$

This generalization is particularly promising because (REVERSE A) can be any list at all, given that A can be any list.

In fact, (3) is easy to prove by induction. Evaluation tells us to induct upon C. The NIL-case evaluates to T, and the inductive step goes through just as in the previous example. So, after a generalization and a second induction, we have proved the NIL-case for (1).

We now proceed to the inductive step. We will assume (1) as our induction hypothesis, and try to prove:

```
(4)       (EQUAL (APPEND (REVERSE A) (REVERSE (CONS B1 B)))
                 (REVERSE (APPEND (CONS B1 B) A))).
```

Evaluation of (4) yields:

```
(5)       (EQUAL (APPEND (REVERSE A) (APPEND (REVERSE B) (CONS B1 NIL)))
                 (APPEND (REVERSE (APPEND B A)) (CONS B1 NIL))).
```

We must now use our induction hypothesis, (1).

The hypothesis tells us that

```
          (APPEND (REVERSE A) (REVERSE B))
```

is equal to:

```
          (REVERSE (APPEND B A)).
```

Although that equality does not occur in the evaluated conclusion, the second of the two terms equated does occur. This is because the term to which it gave rise in the conclusion managed to recurse back to an expression involving the hypothesis term. We can thus "use" the hypothesis by substituting the left-hand side of it for the right-hand side of it in the conclusion. This is called 'cross-fertilization' and produces:

```
(6)       (EQUAL (APPEND (REVERSE A) (APPEND (REVERSE B) (CONS B1 NIL)))
                 (APPEND (APPEND (REVERSE A) (REVERSE B)) (CONS B1 NIL))).
```

As we did with the NIL-case, we now notice that (6) has (three) common subterms "across" the EQUAL. These are the terms (REVERSE A), (REVERSE B), and (CONS B1 NIL). If we generalize as before, replacing these three terms by new skolem constants, C, D, and E, we get:

```
(7)       (EQUAL (APPEND C (APPEND D E)) (APPEND (APPEND C D) E)).
```

But this is just the associativity of APPEND, which was our first example. Thus, we know that a final induction, this time on C, completes the proof.

CHAPTER 3 DESCRIPTION OF THE THEOREM PROVER

3.1 Overview and Control Structure

Several facts are clear from the preceding examples. Evaluation should be tried on the theorem first, since some theorems -- especially the NIL-cases of induction arguments -- will yield to evaluation. As it turns out, it is often helpful to apply normalization and simplification rules to the theorem as well. If these routines do not prove the theorem, cross-fertilization and generalization should be tried. Finally, induction should be resorted to.

In the course of proving a theorem, it may be necessary to use induction several times to establish lemmas (as in the example in Section 2.3). In addition, quite complicated conjunctions and implications often arise (as a result of induction) and must be established. While these requirements could be met by a hierarchical control structure and a data base of the relevant hypotheses, a much more elegant solution is available here. This is explained below.

There are several major routines in the theorem prover, including the evaluation routine, _eval_, which has already been informally discussed. The other routines will be described briefly below. The control structure is then presented, and then each of the major routines is described in detail in the succeeding sections.

normalize is a theorem prover which applies about a dozen rewrite rules to put LISP expressions in a normal form. 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 simply A, as a result of normalization.

<u>reduce</u> is a theorem prover which attempts to propagate the results of the tests in conditional statements down the branches of the COND "tree". Thus, if some expression occurs as the first argument to a COND, it is assumed to be non-NIL on the true-branch and NIL on the false-branch. For example,

```
(COND A (COND A B C) (FOO A))
```

reduces to:

```
(COND A B (FOO NIL)).
```

The three routines <u>eval</u>, <u>normalize</u>, and <u>reduce</u> are often used in sequence until the theorem can no longer be rewritten by any of them. This process is called 'normalation'.

<u>fertilize</u> is responsible for cross-fertilization and using the induction hypothesis when it is an equality.

<u>generalize</u> is responsible for generalizing the theorem to be proved by replacing some common subexpressions by new skolem constants. However, generalization is dangerous (because the generalized version may not be a theorem) and <u>generalize</u> attempts to discover whether the value of the expression being replaced is of a highly constrained type. It does this by automatically writing a new recursive LISP function which is designed to recognize lists in the range of the replaced term. This is done by a routine called <u>typeexpr</u>. This is a good example of using automatic programming to help construct a proof.

Finally, <u>induct</u> is responsible for generating induction formulas. This is done by using <u>eval</u> to determine likely candidates to induct upon, and to determine the precise form of the induction formula.

The control structure of the program is embedded in the syntax of the theorem being proved, and in the routines _eval_, _normalize_, and _reduce_.

For example, if p and q are both boolean expressions and both must be proved, then the theorem becomes (COND p q NIL). This represents the conjunction of p and q in LISP. The two conjuncts are worked on simultaneously by the various rewrite rules.

If p is eventually rewritten to T, the theorem is (COND T q NIL), which _eval_ transforms to q. If q is shown to be T first, the theorem is (COND p T NIL), which _normalize_ rewrites to p. If q is shown to be the same as p, the theorem is (COND p p NIL), which _normalize_ rewrites to simply p.

If it must be shown that p implies q, where p and q are again boolean, the theorem becomes (COND p q T). Again various rewrite rules apply.

All of the logical connectives have COND representations (see the definitions of AND, OR, NOT, and IMPLIES in Appendix A). The theorem prover's logical behaviour is entirely determined by the form of the theorem, and in particular, the kinds of conditional statements present. There is no special logical language. The routines that manipulate LISP expressions serve the double role of manipulating logical ones as well.

As an example, suppose the theorem to be proved is p(A,B), and it is necessary to try induction on A, with induction hypotheses about the CAR and CDR of the term inducted upon. Then _induct_ constructs the expression:

```
(AND p(NIL,B) (IMPLIES (AND p(A1,B) p(A,B)) p((CONS A1 A),B))),
```

which becomes the theorem to be proved. This just normalates to:

```
(COND p(NIL,B)
     (COND p(A1,B)
          (COND p(A,B) p((CONS A1 A),B) T)
          T)
     NIL).
```

(In fact, _induct_ uses the non-primitives AND and IMPLIES only to make its output more readable for the user. _eval_ immediately replaces the non-recursive AND and IMPLIES function calls by their definitions, and _normalize_ cleans the result up, so that the theorem is put into the form above.)

Should p(NIL,B) become T, then the next time the expression is evaluated, it is rewritten to:

```
(COND p(A1,B)
     (COND p(A,B) p((CONS A1 A),B) T)
     T).
```

eval has effectively shifted the attention of the entire system to the induction step.

An outline of the program is given below. The identifier _thm_ is the expression representing the theorem and must be shown to be T.

```
loop:  set oldthm to thm.

       set thm to reduce(normalize(eval(thm))).

       if thm is T, exit.

       if thm is not identical to oldthm, go to loop.

       if fertilization is possible,
            set thm to fertilize(thm)

       otherwise, if thm is of the form (COND p q NIL),
            set thm  to (COND induct(generalize(p)) q NIL)

       otherwise, set thm to induct(generalize(thm)).

       go to loop.
```

The check for theorems of the form (COND p q NIL) (and also of the form (COND p NIL q)) is so that if a conjunction must be proved by induction, the first conjunct, p, is worked on first. Should that succeed, the theorem will be rewritten by the COND evaluation rules and become just the second conjunct.

Thus, the system contains no special logical language or control structure exclusively for implementing the logical connectives. These connectives are all simulated by COND (and the user can use the non-primitives AND, OR, NOT, and IMPLIES which expand into the the proper conditionals). The system has a lot of built-in information about COND. 'Logical operations' are performed by the same routines that simplify LISP expressions. LISP is thus the only knowledge domain involved. The result is that the program is concise, simple, and powerful.

Routines designed for one purpose, say evaluating expressions, can apply their knowledge to other purposes, such as recognizing solved sub-goals and shifting the attention of the system. There are no communications or interface problems preventing unforseen applications of knowledge or serendipity. Finally, such a control structure means that any programming effort devoted to expanding the program's knowledge of the LISP primitives equally benefits the program's logical behaviour and vice versa.

The major routines in the above scheme and the normalation process will now be described in detail.

3.2 Evaluation

There are two useful ways to look at the _eval_ routine. One is as a theorem prover which applies logically sound rewrite rules to its input to produce its output. The other is as a symbolic interpreter for this LISP subset.

We will discuss the theorem prover view first, and for simplicity we will initially ignore non-primitive functions. The view of _eval_ as an interpreter will emerge from this discussion.

eval accepts as input a term in the theory and yields as output an equivalent term. The output is obtained from the input by exhaustively applying the rewrite rules suggested by the following theorems:

```
(CAR NIL) = NIL.

(CDR NIL) = NIL.

(CAR (CONS X Y)) = X.

(CDR (CONS X Y)) = Y.

X = Y -> (EQUAL X Y) = T.

X ≠ Y -> (EQUAL X Y) = NIL.

(EQUAL (CONS X Y) (CONS U V)) = (COND (EQUAL X U)
                                      (EQUAL Y V)
                                      NIL).

(COND (CONS X Y) U V) = U.

(COND NIL U V) = V.
```

That the above formulas are theorems is clear from the non-logical axioms.

For those theorems above which are simple equalities, the corresponding rewrite rule simply replaces any instance of the left-hand term occurring in the input by the appropriate instance of the right-hand term.

In the case of the two theorems:

$$X = Y \rightarrow (\text{EQUAL X Y}) = T.$$

$$X \neq Y \rightarrow (\text{EQUAL X Y}) = \text{NIL}.$$

any instance, (EQUAL x y), of (EQUAL X Y) is replaced by T if x is known to be equal to y, and is replaced by NIL if x is known to be not equal to y.

The routine used to detect whether x and y are equal or not is called <u>ident</u>. This routine returns one of three results: "equal", "unequal", or "unknown", based only on the syntax of x and y. It can be viewed as a (very incomplete) theorem prover for theorems of the form x = y. It returns "equal" if it proves the theorem, "unequal" if it proves the negation of the theorem, and "unknown" if it fails to do either. The routine knows the abbreviation conventions used, and returns "equal" only when x and y are syntactically identical (modulo the abbreviations). The theorem it uses to determine that x and y are unequal is: no explicit list is equal to any of its sublists. This theorem has a trivial inductive proof. The most obvious instance of the theorem is that NIL is never equal to any CONS. Below are some examples of <u>ident</u>'s behaviour.

x	y	<u>ident</u>(x,y)
(CONS A B)	(CONS A B)	equal
(CONS 0 NIL)	T	equal
(CONS A B)	NIL	unequal
(CONS A B)	A	unequal
(CONS A B)	C	unknown
(CONS A B)	(CONS A C)	unknown

The rewrite rules applied by *eval* are given below. Any term of the form of one of those on the left-hand side of an arrow is replaced by the corresponding term on the right-hand side. Any condition on the use of a rule is stated. The rules are applied to all subexpressions until no further rules apply. The rules are justified by the theorems listed above.

```
(CAR NIL) => NIL.

(CDR NIL) => NIL.

(CAR (CONS x y)) => x.

(CDR (CONS x y)) => y.

(EQUAL x y) => T, if ident(x,y) = "equal".

(EQUAL x y) => NIL, if ident(x,y) = "unequal".

(EQUAL (CONS x y) (CONS u v)) => (COND (EQUAL x u)
                                       (EQUAL y v)
                                       NIL).

(COND (CONS x y) u v) => u.

(COND NIL u v) => v.
```

A few examples of evaluation may clarify the definition.

x	*eval*(x)
(CONS A B)	(CONS A B)
(CAR (CONS A B))	A
(CONS (CAR (CONS A B)) NIL)	(CONS A NIL)
(EQUAL 3 (CONS NIL 2))	T
(COND (EQUAL (CDR (CONS A B)) B) (CAR (CONS A B)) NIL)	A
(EQUAL (CONS A B) (CONS C B))	(COND (EQUAL A C) T NIL)
(COND A (CDR (CONS A B)) C)	(COND A B C)

If evaluation yields x as the output for the input term y, we call x the 'value' of y, or say that y 'evaluates to' x. We say that a term is 'unchanged by evaluation' if none of the rules apply.

We are assured that a term and its value are always equal, since the rules applied always replace terms by equal terms. We are also assured that if x is the value of y, then x itself is unchanged by evaluation. This is because to be the output of eval, no further rewrite rules are applicable to x.

We should observe an important feature of evaluation. If the expression evaluated contains no skolem constants, its value will be a specific list, that is, a term composed only of CONS and NIL expressions. The inductive proof of this is straightforward:

If the term has no arguments it is NIL and its value is NIL. If the term has arguments, assume they have been completely evaluated and that they are specific lists. Then the statement holds if the function symbol of the term is CAR or CDR since the output is just a subexpression of the argument. It holds if the function symbol is CONS since no rules apply and the output will be just CONS applied to the two arguments. However, CONS applied to two specific lists is a specific list. The statement holds if the function symbol is EQUAL because we can easily decide if two specific lists are identical, so the output is T or NIL. Finally, it holds if the function symbol is COND because the first argument is either NIL or an explicit CONS, so the value is one of the other arguments (and both are specific lists).

In all of these cases no further evaluation is possible since the output has either been previously evaluated, or is T or NIL. We thus see that this procedure will transform any totally primitive expression (containing no skolem constants) into a specific list. In particular, eval will decide the truth or falsity of any boolean statement involving no skolem constants.

From the form of this inductive argument we derive the basic form of the recursive implementation of eval: evaluate the arguments of any expression and then apply one of the rewrite rules to the top-level expression. The theorem just proved and this implementation suggest the other view of eval, that is, that it is an interpreter for this LISP subset.

The examples exhibited above show that if skolem constants are involved it is possible for the output of eval to be other than a specific list. In fact, it need not even be an explicit list.

We will now consider non-primitive functions and the way evaluation treats them. The obvious rewrite rule is:

$(f\ x_1 \ldots x_n) \Rightarrow \text{defn}(x_1, \ldots, x_n)$, where the defining axiom for f is:
(f X1 ... Xn) = defn(X1, ... , Xn).

This rule is adequate as long as the x_i are specific lists. However, consider what such a rule would allow us to do with (APPEND A B). Applying the rule the first time gives:

```
(COND A
     (CONS (CAR A) (APPEND (CDR A) B))
     B).
```

We can now apply the rule again (to the APPEND inside) and get:

```
(COND A
     (CONS (CAR A)
          (COND (CDR A)
               (CONS (CAR (CDR A)) (APPEND (CDR (CDR A)) B))
               B))
     B).
```

We could expand it again, but clearly we are losing.

We want to halt such an expansion when it is pointless to continue. (In the above example, it was pointless to even start.) This is impossible for arbitrary recursive functions. However, for a large class of functions, including most of those in elementary list processing, a relatively simple restatement of the rule suffices:

```
(f x₁ ... xₙ) => defn(x₁, ... , xₙ) where the
                 defining axiom for f is:
                 (f X1 ... Xn) = defn(X1, ... , Xn),
                 and no new explicit CAR or CDR terms
                 appear in the value of any argument
                 to any recursive call of f in
                 defn(x₁, ... , xₙ).
```

Inspecting the first expansion of (APPEND A B) above we see that (CDR A) appears as the value of an argument to a recursive call of APPEND, and hence, the expansion is not allowed.

Intuitively, this restriction is reasonable. If we are recursing on an argument we usually obtain the value of the argument for the next recursive call by applying some combination of CARs and CDRs to the current value. If we cannot fully evaluate the new arguments, the recursive calls within the definition are, in some sense, more complicated than the original expression.

Formally, the restriction does not affect the validity of the rule. The expanded definition is always equivalent to (f x_1 ... x_n), so it is certainly sound to apply the rule. The restriction merely prevents some applications.

It is easy to see that the following evaluations are permitted under the restriction:

x	eval(x)
(APPEND (CONS A1 A) B)	(CONS A1 (APPEND A B))
(REVERSE (CONS A1 A))	(APPEND (REVERSE A) (CONS A1 NIL))
(MEMBER A (CONS B1 B))	(COND (EQUAL A B1) T (MEMBER A B))
(COPY (CONS A1 A))	(CONS (COPY A1) (COPY A))
(ORDERED (CONS A1 A))	(COND (LTE A1 (CAR A)) (ORDERED A) NIL)

It should be noted that this rewrite rule, even with its restriction, allows evaluation to transform any expression, (f x_1 ... x_n), where f is any total, recursive function, into a specific list, provided the x_i are specific lists. The proof of this, for say, primitive recursive f, is an inductive proof similar to the one that f will always terminate. The proof relies on the fact that once the non-primitive function calls in the definition have been evaluated to specific lists (by the induction hypothesis) the previous theorem applies. The rule may always be applied in these cases (that is, the restriction does not prevent its use) since CAR and CDR of specific lists always evaluate to specific lists (rather than terms with explicit CAR or CDR terms in them).

We thus observe that eval is a symbolic interpreter for this LISP subset in the following sense. Let x be an expression in the theory such that a traditional LISP interpreter (respecting the conventions of this subset) yields some S-expression, s, given x. Then eval(x) is a specific list, l, with the same structure as s. By "the same structure" is meant: l is NIL if and only s is NIL, and l is (CONS y z) if and only if s is (u . v), where y has the same structure as u, and z has the same structure as v.

However, this interpreter is somewhat more flexible than traditional ones in being able to handle skolem constants and expressions containing them.

Evaluation can be used to generate a case analysis for the theorem being proved. For example, consider the expression:

```
(MEMBER A (CONS B1 B)).
```

Evaluation of this expression yields:

```
(COND (EQUAL A B1) T (MEMBER A B)).
```

This just says that the input expression is T if A is equal to B1, or if A is a MEMBER of B. Thus, evaluation has broken up the original expression into two cases.

Notice also that we can discover the kinds of structures a recursive function needs in its recursive arguments, by analyzing the way evaluation halts the expansion of the expression. This provides us with the information necessary to generate induction formulas.

For example, in evaluating (APPEND A B) we noticed that the only "culprit" preventing application of the recursive function rewrite rule was (CDR A). Hence, if the first argument had been a single CONS, evaluation would have proceeded one full step (one application of the recursive function rule). Furthermore, the resulting value would involve APPEND applied only to the CDR of the supplied CONS, so that an induction hypothesis about the CAR would not be used.

In evaluating (COPY A), we find (CAR A) and (CDR A) as culprits. Thus, a single CONS is again sufficient, but one full recursive step leaves an expression containing COPY applied to both the CAR and the CDR. Thus, induction hypotheses about both may be necessary.

We will discuss the details later. Here it is sufficient merely to point out that in order to set up the induction formula it is essential to know why evaluation was halted. In the implementation of eval we will provide mechanisms for communicating this information.

By now the implementation of eval should be obvious. The routine uses an association list to bind the variables in function definitions to their values.

The atoms NIL, skolem constants, T, and the integers evaluate to themselves. All other atoms are evaluated by looking up their values on the association list.

CAR and CDR expressions are handled by evaluating the argument and returning the appropriate component (or NIL) if it is an explicit list. However, if it is not, the input expression with the argument evaluated is returned. In this case there is the possibility that the CAR or CDR expression occurred as an argument to a non-primitive function currently being expanded by the function application mechanism. If this is so, the "failure" of the CAR or CDR indicates that the expansion cannot be allowed. To communicate this information the term is added to a list which is used by the function application mechanism to construct a description of why the function failed to properly expand. This will be explained shortly.

CONS expressions are handled by simply returning the input expression with the two arguments evaluated.

EQUAL expressions cause their two arguments to be recursively evaluated and then compared using ident. If they are identical, eval returns T, if they are definitely unequal, eval returns NIL. If they are both explicit CONS expressions (and neither of the above conditions hold), eval constructs the COND expression indicated by the third EQUAL evaluation rule, and passes that expression to the COND section of eval. If none of the conditions hold, the expression with its arguments evaluated is returned.

COND statements are evaluated by initially evaluating only the first argument. If its value is identical to NIL, the third argument is evaluated and returned. If the value of the first argument is an explicit CONS, the second argument (of the input COND expression) is evaluated and returned. Otherwise, the input expression with all of its arguments evaluated is returned.

Finally, if the function symbol is a non-primitive, the expression is (very carefully) evaluated as follows: The arguments are evaluated recursively. Should any of them have CAR or CDR failures as described above, and should the input expression represent a recursive call of a function whose definition is being evaluated, then the evaluation halts. eval returns the input expression with its arguments evaluated, rather than applying the definition again. If the arguments evaluate without such failures, the definition is retrieved (from the property list of the function symbol) and the variables in it are bound on the association list to the evaluated arguments. Then the definition is evaluated. When completed, the routine determines whether any recursive calls were halted as above. If so, the application of the definition cannot be permitted, and the input expression with its arguments evaluated is returned. In this case, eval constructs and stores a 'fault description' from information collected during the evaluation of the definition.

This is described below. Lastly, if the definition was evaluated without any recursive calls being halted by failed CAR or CDR terms, the evaluated definition is returned.

Except for the allowance for skolem constants, which make certain expressions "unevaluatable", it should be clear that this routine is just a LISP interpreter for this subset.

The fault descriptions constructed and stored by the function entry mechanism are collected at each level in such a way that when the process is completed, the variable _analysis_ contains a list of the fault descriptions constructed for the expressions appearing in the input. There is thus a description for each non-primitive expression which failed to take at least one recursive step.

Each fault description explains why a particular expression (now unknown) failed to recurse. These descriptions are composed of two lists: a bomb list, and a failures list.

A bomb list is a list of pockets, which are lists of failed CAR and CDR terms associated with some recursive call of the function concerned. All of the terms in a pocket were simultaneously recursed upon by the function in that call. There is a pocket on the bomb list for each halted recursive call of the function. To ensure that the function properly recurses one step, the arguments must allow each CAR and CDR in each pocket to evaluate without failure.

A failures list is a list of failed CAR and CDR terms which occurred in the definition evaluated, but not in argument positions of recursive calls of the function concerned. These failures would have appeared in the evaluated definition had recursion been permitted. An examination of the function ORDERED, for example, illustrates that it is possible for a function to have sufficient information to recurse once, without having sufficient information to properly CAR and CDR its arguments in the non-recursive positions of the definition. Although these failures will not prevent recursion, they are used by the induction machinery to provide as much structure as necessary in the induction conclusion.

Several examples of evaluation and the resulting analysis should help the reader to decode (or code) this description.

The input term (APPEND A B) evaluates to (APPEND A B). The analysis list has only one fault description on it. This description contains a bomb list containing only one pocket. The pocket contains (CDR A). The failures list has the term (CAR A) on it. The configuration of analysis is:

```
(((((CDR A))) ((CAR A)))).
```

The input term (EQUAL (COPY A) A) is unchanged by evaluation. Again, analysis contains only one fault description. The bomb list has two pockets in it (one for each recursive call of COPY in the definition). The pockets contain (CDR A) and (CAR A) respectively. No non-recursive failures occurred, so the failures list is empty. analysis is:

```
(((((CDR A))((CAR A))) NIL)).
```

The input term (OR (LTE A B) (LTE B A)) evaluates to:

```
(COND (LTE A B) T (COND (LTE B A) T NIL)).
```

There are two fault descriptions on analysis, one for each LTE call. The bomb list of each description contains only one pocket, but there are two terms in it, since LTE recurses simultaneously on its two arguments. No non-recursive failures occurred. _analysis_ is:

```
((((CDR A)(CDR B))) NIL) ((((CDR B)(CDR A))) NIL)).
```

Finally, (ORDERED (SORT A)) is unchanged by evaluation. The _analysis_ list contains two descriptions on it, one for ORDERED and one for SORT. The ORDERED description describes attempted recursion on (CDR (SORT A)), and numerous non-recursive failures (because ORDERED compares the CAR of its argument to the CAR of the CDR, using LTE). The SORT term attempted recursion on A and non-recursively accessed the CAR of A. Note that the second level recursive calls of LTE by ORDERED and ADDTOLIST by SORT are not reported in _analysis_ since they do not even enter the picture unless the top-level functions properly recurse. The fault description corresponding to the ORDERED term is:

```
((((CDR (SORT A))))
 ((CAR (CDR (SORT A)))(CDR (SORT A))(CAR (SORT A))(CDR (SORT A)))),
```

and the description for the SORT term is:

```
((((CDR A))) ((CAR A))).
```

It should be clear from these examples that the information in _analysis_ is a collection of straightforward descriptions of "what went wrong" in the evaluation of the expression. The information arises naturally in the course of evaluation of an expression with CAR and CDR failures. _eval_ differs from other interpreters in that it collects this information as it arises and then carries on with its symbolic evaluation.

The importance of _eval_ cannot be overstated. It is an efficient and very natural algorithm that always simplifies its input. (As observed, if no skolem constants are involved, it will decide the truth or falsity of a statement.) It can be used to generate a case analysis for the problem at hand, its descriptions of why it failed provide essential information to the routine which generates induction formulas, and it is primarily responsible for converting the induction conclusion into a statement involving the induction hypothesis.

3.3 Normalization

The normalize routine puts its argument expression into a normal form by applying about a dozen rewrite rules. Before the rules are presented, it is convenient to discuss the notion of a boolean expression.

As introduced in Section 1.1, this is an expression whose value is always either NIL or T. Examples of boolean expressions include (EQUAL A B) and (COND A NIL T). If x is a skolemized expression, then x is boolean if and only if (BOOLEAN x) = T is a theorem, where the definition of BOOLEAN is:

```
(BOOLEAN (LAMBDA (X) (COND X (EQUAL X T) T))).
```

The reason this concept is important is that some of the rewrite rules used by normalize come from theorems such as:

```
(BOOLEAN X) = T  ->  (COND X T NIL) = X.
```

Thus, if some expression, x, is known to be boolean, then (COND x T NIL) can be rewritten to simply x.

To effectively implement such rules it is necessary to know when an arbitrary expression is boolean. Ideally, we need a routine which is total and which returns true for input x if and only if (BOOLEAN x) = T is a theorem. But any theorem can be put into this form, because q is a theorem if and only if (COND q T 2) is boolean. Thus, we cannot expect to find the ideal routine above, since it would be a decision procedure for the theory. Instead, we will settle for a routine, called boolean, which is total and which returns true only if its input is boolean. <u>boolean</u> is a theorem prover in its own right, and in order to handle recursive functions, must be capable of inductive arguments.

We say an expression 'is <u>boolean</u>' if <u>boolean</u> returns true for that expression. To determine whether x is <u>boolean</u>, the following procedure is used. If x is an atom, then it is <u>boolean</u> if and only if it is identical to NIL or T. If x is not an atom, consider the possibilities. If the function symbol is CAR or CDR, x is not <u>boolean</u> (since, in general, these are not boolean valued functions). If the function symbol is CONS, x is <u>boolean</u> if and only if it is identical to (CONS NIL NIL). If the function symbol is EQUAL, it is <u>boolean</u>. If the function symbol is COND, x is <u>boolean</u> if and only if both the second and third arguments of the conditional are <u>boolean</u>. Finally, if the function symbol is a non-primitive, x is <u>boolean</u> if and only if the definition of the function is <u>boolean</u>, provided any recursive calls of the function within the definition are (inductively) assumed to be <u>boolean</u>.

Once a non-primitive function has been discovered to be <u>boolean</u> or non-<u>boolean</u>, its property list is so marked. The inductive argument for recursive functions is handled by checking the property list of the function symbol. If known (either way) whether it is <u>boolean</u>, the stored answer is returned. If not known, it is assumed to be <u>boolean</u> by marking the property list. Then the definition of the function is fetched and explored. Any recursive calls are thus automatically assumed to be <u>boolean</u> when the property list is checked. Should the exploration of the definition discover a non-<u>boolean</u> output, the property list entry is reset to non-<u>boolean</u>.

As an example, consider the expression (MEMBER A B). To determine if it is <u>boolean</u> the definition is inspected (since MEMBER is non-primitive). Before doing so however, the <u>boolean</u> property of MEMBER is set to true.

The definition of MEMBER is:

```
(MEMBER (LAMBDA (X Y)
        (COND Y
              (COND (EQUAL X (CAR Y))
                    T
                    (MEMBER X (CDR Y)))
              NIL))).
```

Since the top-level function symbol is COND, the second and third arguments are checked. The third is certainly <u>boolean</u>. The second is a COND, so its second and third arguments are checked. Its second argument is T, and thus <u>boolean</u>. The <u>boolean</u> property is set for the function symbol of its third argument, so it is (inductively) <u>boolean</u>. Hence, (MEMBER A B) is <u>boolean</u>.

The procedure also successfully determines that NUMBERP, and ORDERED (for example) are <u>boolean</u>, and that UNION, LENGTH, and SORT are not.

The procedure will never classify (or 'type') something as <u>boolean</u> when it is not boolean. However, as pointed out earlier, it is possible to contrive examples which are boolean but not <u>boolean</u>.

One such example is of the sort previously mentioned:

```
(COND (EQUAL (TIMES A B) (TIMES B A)) T 2).
```

This is boolean (in fact, it is T), but is typed as non-<u>boolean</u> because 2 is not boolean. The routine makes the assumption that either branch of a COND can be the value of the expression. This is equivalent to assuming (reasonably) that the test is not a theorem. Another sort of example is:

```
          (CAR (LENGTH A)),
```

which is always NIL, and hence, boolean. It is typed as non-boolean because CAR is not, in general, a boolean valued function. Since the routine ignores the values of the arguments to functions other than COND, such subtleties are lost.

These shortcomings are not really serious. In the context in which boolean is used, they can never introduce unsoundness, and experience has shown that such counterexamples seldom arise in real theorems.

Actually, boolean is a specialization of a much more general routine, called typeexpr. This routine writes a LISP function which attempts to recognize the range of the input expression to typeexpr. Given (MEMBER A B) typeexpr could generate the function MEMBTYPE:

```
          (MEMBTYPE (LAMBDA (X) (COND X (EQUAL X T) T))),
```

which is just BOOLEAN. More generally, typeexpr can generate recursive 'type functions'. The routine is explained in Section 3.7.

We now return to the normalization routine itself. The rewrite rules it applies are justified by the following theorems.

```
          (BOOLEAN X) = T  -> (EQUAL X T) = X.

          (EQUAL (EQUAL X Y) Z) -> (COND (EQUAL X Y)
                                         (EQUAL Z T)
                                         (COND Z NIL T)).

          (BOOLEAN X) = T  -> (COND X T NIL) = X.

          (COND X Y Y) = Y.
          (COND X X NIL) = X.

          (COND (COND X Y Z) U V) = (COND X
                                          (COND Y U V)
                                          (COND Z U V)).
```

The last theorem is just an instance of the schema:

```
          (f X1 ... (COND Y U V) ... Xn) = (COND Y
                                                 (f X1 ... U ... Xn)
                                                 (f X1 ... V ... Xn)).
```

As usual, the proofs of these theorems are completely trivial. For example, in the first theorem, X is boolean if it is T or NIL. Assuming that it is T, we must show that (EQUAL T T) = T, which is true. Assuming it is NIL, we must show that (EQUAL NIL T) = NIL, which is true.

The fourth theorem,

```
          (COND X Y Y) = Y,
```

is interesting. In the theory, X must be NIL or some CONS. In both cases, the COND expression is equivalent to Y. In real LISP, X may fail to terminate, so that the left-hand side never terminates when evaluated, while the right-hand side might. Of course, if the left-hand side does terminate, it is equal to Y.

The general distribution schema is also interesting. Its proof is not, since it is a straightforward case analysis: Y = NIL, or Y = (CONS (CAR Y) (CDR Y)).

In addition, normalization relies on four theorems used by _eval_. These are the theorems that evaluate EQUAL if the two arguments are of known relationship, and the theorems that evaluate COND when the first argument is an explicit list. This redundancy is commented on in Section 6.3.

The rewrite rules themselves are given below. Their correspondence with the theorems above should be obvious. Any rule involving EQUAL has a symmetric version not presented -- that is, the order of the arguments in an EQUAL is not critical. Since some expressions may be rewritten by two or more rules, the order in which they are listed is the order in which they are preferred. Finally, <u>normalize</u> recursively normalizes the argument expressions to any input expression before any of these rules are applied to the top-level expression.

(EQUAL x y) => T, if <u>ident</u>(x,y) = "equal".

(EQUAL x y) => NIL, if <u>ident</u>(x,y) = "unequal".

(EQUAL x T) => x, if x is <u>boolean</u>.

```
(EQUAL (EQUAL x y) z) => (COND (EQUAL x y)
                               (EQUAL z T)
                               (COND z NIL T)).
```

(COND (CONS x y) u v) => u.

(COND NIL u v) => v.

(COND x T NIL) => x, if x is <u>boolean</u>.

(COND x y y) => y.

(COND x x NIL) => x.

```
(COND (COND x y z) u v) => (COND x
                                 (COND y u v)
                                 (COND z u v)), where either
                           y or z is NIL, or u and v are
                           not NIL.
```

```
(f x_1 ... (COND y u v) ... x_n) => (COND y
                                          (f x_1 ... u ... x_n)
                                          (f x_1 ... v ... x_n)),
                                    where f is not COND.
```

The rather complicated restriction on rewriting

(COND (COND x y z) u v)

causes conjunctions to be treated delicately. For example, if v is NIL, then the expression represents the conjunction of (COND x y z) and u. In this case, the first conjunct is not to be distributed over the second (since it is preferable to work on conjuncts separately). However, if z is NIL, then the first conjunct is itself a conjunction. The rule is then allowed to rewrite the expression; thus

(x & y) & u

is rewritten to:

x & (y & u).

The basic philosophy of <u>normalize</u> is to eliminate EQUAL and COND when possible. Compound EQUAL statements are broken into conditionals so that the cases can be handled separately. COND statements are distributed over each other and other function symbols, provided conjunctions are left intact.

Note that the fact that some boolean expressions are not <u>boolean</u> does not introduce unsoundness. It merely means that two rules cannot be applied.

Since the rules always replace terms by equivalent ones, the output of normalize is equivalent to its input.

Below is an example of normalization. The function GT is <u>boolean</u> (as can be confirmed by inspecting its definition in Appendix A). The expression:

```
(COND (COND (COND A B A) (COND B T NIL) T)
      (COND (EQUAL A (COND (GT B B) A (CONS A1 A))) NIL T)
      (COND A1 NIL NIL))
```

normalizes to:

```
(COND (COND A
            (COND B (COND B T NIL) T)
            (COND A (COND B T NIL) T))
      (COND (GT B B) NIL T)
      NIL).
```

3.4 Reduction

The reduce routine is responsible for propagating the results of the tests in conditional statements down the branches of the conditional tree. For example, in an expression of the form:

```
(COND x p(x) q(x)),
```

we can assume that any x occurring in p(x) is non-NIL, and any x in q(x) is NIL.

This is important because normalization often produces expressions in which p(x) and q(x) involve further conditional statements with x as the test. These redundant conditionals can be pared away, and this is what reduce does.

The preceding example of normalization illustrates this. The output of normalize is at the top of the page. Note that both A and B occur as tests in conditionals in the branches of conditionals with A and B as tests. Using reduce ,the normalized expression from the last section can be transformed to:

```
(COND (COND A (COND B T T) T)
      (COND (GT B B) NIL T)
      NIL).
```

(This expression can be normalized again now, yielding:

```
(COND (GT B B) NIL T).)
```

Now that the need for reduction is clear, the details will be presented.

reduce relies on three equality schemas. The first is:

```
(COND X p(X) q(X)) = (COND X p((CONS (CAR X) (CDR X))) q(NIL)).
```

This is seen to be an equality by considering the cases: X = NIL, or X = (CONS (CAR X) (CDR X)).

The second equality is derived from the first, with the additional hypothesis that X is boolean. In this case, when X = (CONS (CAR X) (CDR X)) then X = T. Thus:

```
(BOOLEAN X) = T  -> (COND X p(X) q(X)) = (COND X p(T) q(NIL)).
```

Finally, the third equality is based on the fact that if X is an equality statement, then when it is non-NIL the two terms equated can be assumed to be equal, and one can be substituted for the other:

```
(COND (EQUAL X Y) p(X) q((EQUAL X Y))) =
(COND (EQUAL X Y) p(Y) q(NIL)).
```

In essence, reduce selectively applies these three equality schemas to all subexpressions of its input and then uses the COND evaluation rules to eliminate any redundant tests. That is, if x is tested in p(x) as above, then when p(x) is replaced by p((CONS (CAR x) (CDR x))) or p(T), then the COND evaluation rules remove all tests on x, since it is now an explicit list. The reason the adverb "selectively" is used above is that reduce chooses to replace by (CONS (CAR x) (CDR x)) only those x which occur as the first argument of conditionals in p(x), rather than all occurrences of x in p(x). In addition, the third equality can be used indefinitely, since the right-hand side is still of the form of the left-hand side. Thus, this rule is used carefully.

It is probably best to describe the algorithm for reducing an expression. The routine uses substitution to replace x by T or NIL when possible. When x is not _boolean_, it uses an "assumption list" to remember that x is being assumed non-NIL on the true-branch (rather than actually replacing x by (CONS (CAR x) (CDR x))).

To _reduce_ an expression, z, with assumption list, _a_, the following procedure is used:

If z is an atom, return z. If z is of the form (f z_1 ... z_n), where f is not COND, _reduce_ each of the arguments independently. That is, return:

(f _reduce_(z_1,_a_) ... _reduce_(z_n,_a_)).

Otherwise, z is of the form (COND x p(x) q(x)). Consider the possible forms of x.

If x is a conditional statement, first _reduce_ each of the arguments independently, then rematch the COND pattern shown above and continue as below.

If x is an explicit CONS or is a member of the assumption list, _a_, return:

reduce(p(x),_a_).

If x is NIL, return:

reduce(q(x),_a_).

If x is non-_boolean_, _reduce_ the true-branch after assuming x is non-NIL by adding it to _a_, and _reduce_ the false-branch after replacing x by NIL. That is, return:

(COND x _reduce_(p(x),_cons_(x,_a_)) _reduce_(q(NIL),_a_)).

If x is an equality of the form (EQUAL u v), where v is a specific list, let r be p(x) with all occurrences of u replaced by v, and return:

(COND (EQUAL u v) _reduce_(r,_a_) _reduce_(q(NIL),_a_)).

Finally, if x is a boolean expression other than an equality as above, return:

(COND x reduce(p(T),a) reduce(q(NIL),a)).

Henceforth, the abbreviation 'reduce(x)' will mean reduce(x,NIL). The 'reduction' of x is the output of reduce(x).

This algorithm conservatively applies the three reduction equalities noted earlier and evaluates any COND statement with an explicit list as its first argument. Thus, the reduction of x is equivalent to x.

In applying the third reduction equality, the algorithm requires that the v in (EQUAL u v) be a specific list. Of course, the symmetric rule is included as well. At most one of the two terms, u or v, will be a specific list, since if both were specific, the EQUAL expression could have been evaluated or normalized to T or NIL. The requirement that one be specific is to prevent the indefinite application of the rule to its own output. Substituting v for u in p(x) completely eliminates u from the true-branch without losing any information about it. The fertilization routine is responsible for more sophisticated equality substitutions.

As an example of reduction, the expression:

```
(COND A
      (COND (MEMBER A B)
            (COND (MEMBER A C) (MEMBER A B) (COND A T NIL))
            (COND A T (MEMBER A C)))
      (COND (EQUAL B1 A)
            (COND B1
                  (COND A T NIL)
                  (COND (EQUAL B1 A) T (EQUAL B1 C)))
            (COND (EQUAL A C)
                  (COND A NIL T)
                  (COND (EQUAL B1 A) NIL T)))),
```

reduces to:

```
(COND A
      (COND (MEMBER A B)
            (COND (MEMBER A C) T T)
            T)
      (COND (EQUAL B1 NIL)
            (COND (EQUAL NIL NIL) T (EQUAL NIL C))
            (COND (EQUAL NIL C) T T))).
```

(The expression above normalizes to T.)

3.5 Normalation

The process of applying eval, normalize, and reduce to an expression until it can no longer be rewritten by these routines is called 'normalation'. Normalation preserves equality, since each of the basic routines yields output equivalent to the input. Experience shows that normalation is an efficient and fairly complete theorem prover for list theory without induction. If a skolemized expression normalates to T then the formula represented by the expression is a theorem.

In order for the theorem prover to prove a theorem, the theorem must normalate to T, or else fertilization, generalization, or induction must produce a new expression which implies the original one and which normalates to T. This process is thus the "central processor" of the theorem prover.

An example of normalation is given below. Consider the expression:

```
(AND (EQUAL (APPEND NIL (APPEND B C)) (APPEND (APPEND NIL B) C))
     (IMPLIES (EQUAL (APPEND A (APPEND B C))
                     (APPEND (APPEND A B) C))
              (EQUAL (APPEND (CONS A1 A) (APPEND B C))
                     (APPEND (APPEND (CONS A1 A) B) C)))).
```

This is just the induction formula produced for the theorem:

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

Evaluation of this formula yields:

```
(COND (COND (EQUAL (APPEND A (APPEND B C))
                   (APPEND (APPEND A B) C))
            (COND (EQUAL (APPEND A (APPEND B C))
                         (APPEND (APPEND A B) C))
                  T
                  NIL)
            T)
      T
      NIL),
```

which normalizes to:

```
(COND (EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C))
      (EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C))
      T),
```

and reduces to:

```
(COND (EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C))
      T
      T).
```

This normalizes to T, proving the associativity of APPEND.

In addition to ultimately transforming the theorem into T (when the program wins), normalation simplifies expressions so that additional appeals to induction, or to the routines _fertilize_ and _generalize_, can produce expressions which do normalate to T.

For example, the theorem:

```
(IMPLIES (AND (GT A B) (GT B C)) (GT A C)),
```

which expresses the transitivity of the function GT (greater than), normalates to:

```
(COND (GT A B) (COND (GT B C) (GT A C) T) T).
```

The induction formula for this theorem is fairly complicated because the program (correctly) inducts upon A, B, and C simultaneously:

```
(AND (AND (COND (GT A NIL) (COND (GT NIL C) (GT A C) T) T)
          (AND (COND (GT NIL B) (COND (GT B C) (GT NIL C) T) T)
               (COND (GT A B) (COND (GT B NIL) (GT A NIL) T) T)))
     (IMPLIES (COND (GT A B) (COND (GT B C) (GT A C) T) T)
              (COND (GT (CONS A1 A) (CONS B1 B))
                    (COND (GT (CONS B1 B) (CONS C1 C))
                          (GT (CONS A1 A) (CONS C1 C))
                          T)
                    T))).
```

However, this formula normalates to:

```
(COND (GT A B) (COND B (COND A T NIL) T) T).
```

Normalation transforms the induction formula for the transitivity of GT into the simpler expression that if A is greater than B, then if B is non-NIL, A is non-NIL. This is then proved by induction -- which is to say, induction generates an equivalent formula which normalation can transform into T.

The ability to transform complex expressions, such as the one above, into equivalent ones, makes normalation very useful in the automatic programming feature of the theorem prover. After one routine has generated a recursive function definition to fit a particular need, normalation is used to optimize the function body. Since the output is equivalent to the input, the new, more efficient definition has the same properties of the less efficient but easier to write definition. Section 3.7 discusses this in detail.

Chapter 4, which contains sample output from the program during four proofs, contains many examples of evaluation, normalization, and reduction.

3.6 Fertilization

As indicated in the example in Section 2.3, fertilization is the process of making equality substitutions.

The basic idea behind fertilization is that if x = y is being assumed, then p(y) is equivalent to p(x). If involved in a proof, p(x) may be easier to work with than p(y) because of properties of x, y, or p.

The general theorem schema upon which fertilization relies is:

```
((BOOLEAN p(Y)) = T  &  (BOOLEAN Z) = T)
                    ->
(COND (EQUAL X Y) p(Y) Z) = (COND (COND p(X)
                                        T
                                        (COND (EQUAL X Y) NIL T))
                                  (COND Z T (EQUAL X Y))
                                  NIL).
```

This means that if an expression of the form:

```
(COND (EQUAL x y) p(y) z),
```

where p(y) and z are boolean, occurs in the theorem, it can be replaced by the equivalent expression:

```
(COND (COND p(x) T (COND (EQUAL x y) NIL T))
      (COND z T (EQUAL x y))
      NIL).
```

Notice that x has been substituted for y in p(y).

In the special case where z is T (that is, the original expression is an implication) the schema simplifies considerably. If p(y) is boolean, then

```
(COND (EQUAL x y) p(y) T)
```

can be rewritten to:

```
(COND p(x) T (COND (EQUAL x y) NIL T)).
```

This is just the disjunction of p(x) and the inequality $x \neq y$. The idea is that the statement 'x = y implies p(y)' is equivalent to 'p(x) or $x \neq y$'. The case where z is not T is just a generalization of this.

Let us consider an example. In proving that TIMES (multiplication defined in terms of addition) is commutative, the induction step is:

```
(IMPLIES (EQUAL (TIMES A B) (TIMES B A))
         (EQUAL (TIMES A (CONS B1 B)) (TIMES (CONS B1 B) A))).
```

This normalates to:

```
(COND (EQUAL (TIMES A B) (TIMES B A))
      (EQUAL (TIMES A (CONS B1 B)) (PLUS A (TIMES B A)))
      T).
```

This is of the form:

```
(COND (EQUAL x y) p(y) T),
```

so fertilization applies, and substitutes x for y in p(y), producing:

```
(COND (EQUAL (TIMES A (CONS B1 B)) (PLUS A (TIMES A B)))
      T
      (*1)),
```

where (*1) is a term known to be equal to:

```
(COND (EQUAL (TIMES A B) (TIMES B A)) NIL T).
```

Shifting to algebraic notation, the original theorem was:

$$A \times B = B \times A.$$

Induction on B and normalation produced the inductive step:

$$A \times B = B \times A \rightarrow A \times (1 + B) = A + (B \times A).$$

This was rewritten by <u>fertilize</u> to:

$$A \times (1 + B) = A + (A \times B) \quad \vee \quad A \times B \neq B \times A.$$

However, the negative equality is "hidden" in (*1). This is done on the grounds that it has been "used". This has a consequence which is not immediately obvious. The (*1) term is never expanded into the above inequality, but remains simply (*1) throughout the rest of the proof. As a result, if a second induction is performed, none of the variables appearing in the term to which (*1) is equal are affected.

In particular, if

$$A \times (1 + B) = A + (A \times B) \quad \vee \quad (*1)$$

is proved by induction on A, then the term (*1) is not altered by the induction since no A (explicitly) occurs in it. It is as if the A in the (*1) term were standardized apart (i.e., renamed), and then the more general theorem:

$$A \times (1 + B) = A + (A \times B) \quad \vee \quad A' \times B \neq B \times A',$$

proved by induction on A.

By hiding the inequality it is protected from further inductions (thereby protecting further inductions from it). The (*1) expression is introduced to keep the rewritten expression formally equivalent to the original one (until induction is used). This allows fertilization to freely replace any such expression in the theorem.

The program is "betting" that p(x) will be T, and has "thrown away" the equality. The heuristic grounds on which this is done is that the equality was "used" in rewriting p(y) to p(x), and its contribution to the proof is now complete.

Often, formulas of the form:

```
(COND (EQUAL x y) p(y) z)
```

arise out of induction, where (EQUAL x y) is from the induction hypothesis, and p(y) is the normalated conclusion. Usually z is T, but it can be a more complex expression "pushed back" into that position by normalation.

The reason it is reasonable to "bet" on p(x) rather than p(y) in these circumstances is that if indeed p(y) is the normalated conclusion, then part of it, namely, y, has been converted into that part of the hypothesis from which it came. The rest of it came from x, which failed to properly recurse.

But if x and y are known to be equal, then by substituting x for y in the conclusion we eliminate y from the theorem (provided we also throw away the used equality) and obtain a new theorem which is entirely of the "genre" of x. Then perhaps something can be done which allows x to properly recurse in induction.

Intuitively, fertilization works because x and y require different inductive approaches in order to properly reappear in the normalated conclusion. The most common difference is that they simply require induction on different terms. The first induction done, for example, on B in the TIMES example above, allows y to recurse. In the example, the (TIMES B A) term reappeared in the conclusion because it recursed on B. However, the (TIMES A B) term did not. Because y properly recursed, it could be eliminated from the theorem. This produced a new theorem involving x and its descendant, (TIMES A (CONS B1 B)), provided the equality was thrown away. A second induction, this time on A, allows the x-like terms to recurse.

If the used equality were not thrown away, the y term would "compete" with the x term in the selection of what to induct upon. In this example, even if the x term "won" and A was inducted upon, the y term would not properly recurse, even though it does contain A. Since it does contain A it would be changed by the induction, and it would therefore fail to match in the conclusion.

The point is that if one is proving a disjunction, p(A) v q(A), by induction, but there is reason to believe that p(A) is a theorem (it is just the conclusion of a previous induction argument, after the induction hypothesis has been used), then proving p(A) might be easier. Furthermore, if there is reason to believe that p and q recurse on different terms (and p recurses on A), then proving the induction step:

p(A) v q(A) -> p((CONS A1 A)) v q((CONS A1 A)),

may be very difficult. In particular, since p(A) recurses upon A, the p term in the conclusion should become r(p(A)), while the q term does not recurse. Thus, one must show:

p(A) v q(A) -> r(p(A)) v q((CONS A1 A)).

In particular, one must deal with the case where the p(A) in the hypothesis is false, and q(A) is true, and show that the conclusion is true. If, on the other hand, we had decided to prove the much stronger theorem:

p(A) -> p((CONS A1 A)),

by ignoring the second disjunct (which is the used equality), the situation is much clearer, because this reduces to:

p(A) -> r(p(A)).

Since there is reason to believe that p(A) is a theorem, and since there is also reason to believe that it is "incompatible" with q(A), it is very useful to ignore q(A) in the induction. This is precisely why the used equality, x = y, is hidden in the (*1) term and the entire effort of the theorem prover devoted to proving the fertilized conclusion.

In the example above, the incompatibility of x and y was due to their recursing on different terms. Another frequent use of fertilization is that it allows a theorem to be generalized (by introducing a common subterm on both sides of an equality, for instance) which in turn allows a second induction to succeed.

Let us return to the example in Section 2.3 in which fertilization was introduced. The theorem to be proved was:

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

Induction on B was used, and the NIL-case was dismissed after a second induction. This left the induction step for the original theorem:

```
(COND (EQUAL (APPEND (REVERSE A) (REVERSE B))
             (REVERSE (APPEND B A)))
      (EQUAL (APPEND (REVERSE A)
                     (APPEND (REVERSE B) (CONS B1 NIL)))
             (APPEND (REVERSE (APPEND B A)) (CONS B1 NIL)))
      T).
```

(The formula above is the induction step after it has been normalated.)

Note that this formula is of the form:

```
(COND (EQUAL x y) p(y) T).
```

Thus, fertilization applies, and rewrites it to:

```
(COND (EQUAL (APPEND (REVERSE A)
                     (APPEND (REVERSE B) (CONS B1 NIL)))
             (APPEND (APPEND (REVERSE A) (REVERSE B))
                     (CONS B1 NIL)))
      T
      (*1)),
```

where (*1) is known to be equal to:

```
(COND (EQUAL (APPEND (REVERSE A) (REVERSE B))
             (REVERSE (APPEND B A)))
      NIL
      T).
```

As noted earlier, if the common subterms in the output of fertilization are replaced by skolem constants, the result is the statement that APPEND is associative (provided the (*1) term is ignored). This theorem has a trivial inductive proof, but it was necessary to generalize the theorem, and the generalization was not possible until the equality hypothesis had been used to introduce the common subterms.

Two of the proofs in Chapter 4 illustrate fertilization. The two theorems are:

(EQUAL (REVERSE (REVERSE A)) A).

(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A))).

In both of the specific examples given in this section the normalated conclusion, p(y), was an equality. This is not always the case, but we will consider it to be so for the moment.

It is possible that y occurs on both sides of the equality in the conclusion. The expression to be fertilized is then of the form:

(COND (EQUAL x y) (EQUAL p_1(y) p_2(y)) z),

where z is boolean. In this case, x is substituted for y only on the right-hand side of (EQUAL p_1(y) p_2(y)). The result is thus:

(COND (COND (EQUAL p_1(y) p_2(x)) T (*n))
      (COND z T (EQUAL x y))
      NIL).

The term which properly recursed is replaced. When two equalities are involved and the fertilization was right-side into left-side or left-side into right-side, it is called 'cross-fertilization'.

Occasionally both x and y occur in the conclusion. Fertilization is used to produce a new term of uniform "genre". Cross-fertilization is preferred, but if both allow cross-fertilization, the smaller of x and y is substituted for the larger.

Finally, it is possible that the evaluated conclusion, p(y), is no longer an equality. If one or both of the two expressions, x, or y, occur in it, fertilization replaces all occurrences of one by the other. Size is used to decide which is replaced if a choice is possible.

It should be obvious that fertilization allows x to be replaced by y under conditions exactly symmetric to those described as allowing y to be replaced by x. The term p(x) is said to have been 'fertilized by (EQUAL x y)'. In all cases, a term such as (*n) is produced and the negated equality is stored on the property list of the atom. In addition, the term is known to be boolean (so that various rewrite rules can still be applied), and it acts like NIL with regard to the COND distribution rule in normalize. (That is, a term such as:

```
        (COND (COND x y z) u (*n)),
```

is not rewritten to:

```
        (COND x (COND y u (*n)) (COND z u (*n))).
```

Instead, it is treated as a conjunct.)

The details of the fertilize routine can now be filled in. It operates on an expression and returns an equivalent one (modulo terms such as (*n)). The routine searches the expression (depth-first) for the first expression of the form:

```
        (COND (EQUAL x y) p(y) z),
```

where y actually occurs in p(y), and p(y) and z are boolean. If such an expression is found, we say 'fertilization applies', and fertilize replaces the COND expression (above) by the equivalent one:

```
        (COND (COND p(x) T (*n)) (COND z T (EQUAL x y)) NIL),
```

where p(x) has been fertilized as described above, and (*n) is a new term known to be boolean, with (COND (EQUAL x y) NIL T) on its property list.

It should be noted that if z is T, fertilize replaces the original COND by the simpler expression:

```
(COND p(x) T (*n)),
```

thereby saving normalize a few steps.

The output of fertilize is a copy of the input expression with the COND expression replaced as described.

Historically, fertilization was performed by the reduction routine. A vestige of it still remains, in the rule that allows substitution of a specific list for a term. Fertilization was removed from reduce to eliminate it from the normalation cycle. Thus, it is performed only after all of the more obvious rewrite rules have been applied.

3.7 Generalization and Type Functions

Often a theorem must be generalized before it can be proved by induction. This is usually expressed by saying that the theorem "is not strong enough to carry itself through induction". The routine generalize is responsible for generalizing the theorem to be proved.

An example of a theorem requiring generalization was given in Section 2.3, and discussed again above. In proving:

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

the induction step led to the theorem:

```
(COND (EQUAL (APPEND (REVERSE A)
                     (APPEND (REVERSE B) (CONS B1 NIL)))
             (APPEND (APPEND (REVERSE A) (REVERSE B))
                     (CONS B1 NIL)))
      T
      (*1)).
```

This must be generalized in order to prove it. If (REVERSE A), (REVERSE B), and (CONS B1 NIL) are replaced by the new skolem constants C, D, and E, the result is:

```
(COND (EQUAL (APPEND C (APPEND D E)) (APPEND (APPEND C D) E))
      T
      (*1)),
```

which is just the statement that APPEND is associative.

A theorem is generalized if it has been normalated and fertilized until these routines no longer apply. generalize finds all of the non-atomic subterms which occur in the theorem on opposite sides of an EQUAL term or in the hypothesis and conclusion of a conditional statement representing an implication. Subject to certain constraints defined below,it replaces these common subterms by new skolem constants.

For example, in proving the theorem that the output of SORT is ORDERED:

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

the induction step normalates to:

```
(COND (ORDERED (SORT A)) (ORDERED (ADDTOLIST A1 (SORT A))) T).
```

generalize detects that this conditional represents an implication and replaces the common subterm, (SORT A), by the new skolem constant GENRL1. The result is:

```
(COND (ORDERED GENRL1) (ORDERED (ADDTOLIST A1 GENRL1)) T),
```

which is one of the primary properties of ADDTOLIST.

This is a generalization because the original theorem is equivalent to:

```
∀X,Y((ORDERED (SORT X)) = T  ->
     (ORDERED (ADDTOLIST Y (SORT X))) = T).
```

The output of generalize is equivalent to the more general:

```
∀Z,Y((ORDERED Z) = T -> (ORDERED (ADDTOLIST Y Z)) = T).
```

This kind of generalization is reasonable on the heuristic grounds that it simplifies the theorem but retains a large degree of the original structure of the theorem. The hope is that the role played by the replaced term(s) is not critical in the proof except insofar as the term occurs on both sides of the EQUAL or implication.

Generalization is dangerous however. The role played by the common subterms may be important and the generalized formula may not be a theorem.

For example, the function CDRN takes two arguments, X and Y, and returns the X^{th} CDR of Y if Y has X elements, and NIL otherwise. The following is a theorem:

```
(COND (CDRN A B) (MEMBER (CAR (CDRN A B)) B) T).
```

This states that if the A^{th} CDR of B is non-NIL, then its CAR is an element of B. We might be tempted to generalize this to:

```
(COND GENRL1 (MEMBER (CAR GENRL1) B) T),
```

which states that if GENRL1 is non-NIL, its CAR is an element of (the random list) B. This is not a theorem.

One might imagine that the problem can be avoided if terms which share variables with the ungeneralized part of the expression are not replaced. This is quite reasonable, but has numerous counter-examples, such as:

```
(EQUAL (PLUS (TIMES B C) (TIMES (TIMES A B) C))
       (TIMES (PLUS B (TIMES A B)) C)).
```

Before this can be proved by induction, (TIMES A B) must be replaced, generalizing the theorem to:

```
(EQUAL (PLUS (TIMES B C) (TIMES GENRL1 C))
       (TIMES (PLUS B GENRL1) C)).
```

(This is taken from the program's proof that TIMES is associative. The example is discussed further in Section 6.2.)

No clear solution has been found for this problem. However, to avoid the problem as much as possible, the program refuses to generalize an expression if it represents the entire hypothesis of an implication. This works well for terms such as the CDRN one, which are tested (against NIL) to determine if their values are relevant before using them.

Another example of a false generalization is in the theorem:

```
(EQUAL (LENGTH (LENGTH A)) (LENGTH A)).
```

This is a theorem since the LENGTH of a list is a number -- that is, a list of NILs as long as the argument list. The theorem can be very neatly generalized to the non-theorem:

```
(EQUAL (LENGTH GENRL1) GENRL1).
```

An elegant solution to this type of problem has been found.

The "proper" generalization of the theorem is:

```
(IMPLIES (NUMBERP GENRL1) (EQUAL (LENGTH GENRL1) GENRL1)),
```

where the definition of NUMBERP is:

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

NUMBERP recognizes numbers. The generalized theorem is that the LENGTH of any number is itself.

The general schema being used here is that if p(r) = T is to be proved, it is sufficient to prove:

```
rtype(X) = T  ->  p(X) = T,
```

and

```
rtype(r) = T.
```

The question then arises: if r is to be generalized, is it possible to automatically generate the expression rtype(X)? In the above example, is it possible to generate NUMBERP, given (LENGTH A)? In a number of cases the program can generate such expressions and functions.

The routine which generates these 'type expressions', rtype(X), is called _typeexpr_. Given an expression, r, possibly involving recursive functions, _typeexpr_ will return a new expression, rtype(X), which is supposed to be equivalent to T only for X in the range of r. Usually in doing so, _typeexpr_ will write and _define_ new, recursive LISP functions. These functions, written by _typeexpr_ to recognize the output of other functions, are called 'type functions'.

The _generalize_ routine may be described as follows. It takes as its argument some expression representing a theorem. It finds any subterm, r, that is non-atomic and which occurs in both arguments of an EQUAL or in the hypothesis and conclusion of an implicational conditional statement, but not occurring as the hypothesis of any such conditional, and not occurring within any larger such common subterm. If no such r is found, _generalize_ returns the original expression.

If such an r is found, let the input expression be p(r). The routine uses _typeexpr_ to generate the type expression, rtype(X), for r. If rtype(X) is just T (that is, the range of r is the entire universe), _generalize_ returns the result of generalizing p(x), where x is a new skolem constant. That is, _generalize_ replaces r by a new skolem constant and then generalizes the remaining r's.

If rtype(X) is not T, generalize returns:

(IMPLIES rtype(x) _generalize_(p(x))),

where x is a new skolem constant.

Thus, _generalize_ replaces every such r by a new constant and uses type expressions and type functions to restrict the constants to certain ranges. The output of the routine becomes the new theorem to be proved.

The fact that rtype(r) = T, which must be true if the new theorem is to imply the old, is true by the method used to construct rtype(X). This will now be described.

Let us return to the particular problem of generating NUMBERP given (LENGTH A). The definition of LENGTH is:

```
(LENGTH (LAMBDA (X)
        (COND X
              (ADD1 (LENGTH (CDR X)))
              0))),
```

where ADD1 is just:

```
(ADD1 (LAMBDA (X) (CONS NIL X))).
```

When the definition of LENGTH is introduced into the system (by _define_) it is normalated to simplify it. This expands it to:

```
(LENGTH (LAMBDA (X)
        (COND X
              (CONS NIL (LENGTH (CDR X)))
              0))),
```

since ADD1 is non-recursive.

We want to generate a function, LENGTYPE, which takes one argument, X, and returns T or NIL according to whether X could be a value of LENGTH applied to anything. If X is to satisfy LENGTYPE, it must be the value of one of the two branches of the COND expression in the (normalated) definition of LENGTH. Thus, X must either be 0, or X must be the value of (CONS NIL (LENGTH (CDR A))). But to be the value of the CONS term, it must be non-NIL, its CAR must be NIL, and its CDR must be the value of some LENGTH expression. But this last condition means (inductively) that (CDR X) must satisfy LENGTYPE.

Thus, we have determined that for X to satisfy LENGTYPE, the following must hold:

$$X = 0 \ \ v \ \ (X \neq NIL \ \& \ (CAR\ X) = NIL \ \& \ (LENGTYPE\ (CDR\ X)) = T).$$

Expressing this in LISP gives:

```
(COND (EQUAL X 0)
      T
      (COND X
            (COND (EQUAL (CAR X) NIL)
                  (LENGTYPE (CDR X))
                  NIL)
            NIL)).
```

By construction, this is the definition of LENGTYPE. However, if we normalate this expression, we find that it is equivalent to:

```
(COND X
      (COND (CAR X) NIL (LENGTYPE (CDR X)))
      T).
```

But this is just the definition of NUMBERP (modulo the name). Thus, we have managed to construct, by a completely general procedure, a recursive function which returns T if and only if its argument is a number. Furthermore, the definition is very efficient within the constraints of the language. The procedure just employed is implemented in the routine <u>typeexpr</u>.

This routine recursively explores an expression, r, and possibly several function definitions, and generates a new expression, rtype(X), and possibly several new LISP functions. This new expression will have X free in it. Ideally, when some new term, x, is substituted for X in rtype(X), the result will be equivalent to T if and only if x is within the range of r. Since, as we saw in the case of boolean, this is in general not possible, certain heuristics will be used which make rtype(X) err on the liberal side. For some x, not within the range of r, rtype(x) will be T. However, for all x within the range of r, rtype(x) will be T. Thus, rtype(r) is T. Since such type expressions will be used to restrict generalizations, this error does not introduce unsoundness; it may however allow the program to make too general a generalization.

We will now describe the routine. In considering expressions with non-primitive top-level function symbols, we will initially ignore the arguments and merely type the function as if its arguments were unrestricted. Let r be the expression being explored, and X be the free variable.

If r is a skolem constant or a variable in a function definition, it can have any value whatsoever, so the output of typeexpr is T.

If r is a specific list, in particular, NIL, T, or a number, return (EQUAL X r).

If r is any other CONS expression, obtain p(X) by recursively finding the type expression for the first argument of the CONS, and obtain q(X) by finding the type expression for the second argument of the CONS. Return the expression:

(COND X (COND p((CAR X)) q((CDR X)) NIL) NIL).

If r is a CAR or CDR expression, assume the output can be anything and return T.

If r is an EQUAL expression, the output is T or NIL, so return:

(COND X (EQUAL X T) T).

If r is a COND expression, assume either branch can be the value of the expression. Ignore the condition tested. Obtain p(X) by finding the type expression for the second argument, and q(X) by finding the type expression for the third argument, and return:

(COND q(X) T p(X)).

Otherwise, r is of the form (f x_1 ... x_n), where f is a non-primitive. If the type function for f is already known to be ftype, return (ftype X). (A function's type function, when known, is stored on its property list.) If its type function is not known, generate a new function name, ftype, and store it on the property list of f as the type function of f. Recursively find the type expression of the definition of f, and call this p(X). If (ftype X) occurs in p(X), replace it by NIL. Define the new type function by:

(ftype (LAMBDA (X) <u>normalate</u>(p(X)))),

and return (ftype X).

Several points should be made about this definition. If r is an application of a non-primitive function, f, then a new recursive function, ftype, is written and defined. This is done by creating the type expression for the definition of f. Any expression of the form (ftype X) in this type expression is replaced by NIL. If this were not done, ftype would not be a total function. The reason (ftype X) is replaced by NIL (rather than something else) is as follows: (ftype X) can only be introduced if a recursive call of f appears as one of the branches in a COND statement in the definition of f. If X was the output of this recursive call of f, then it was also the output of some other exit of f. Hence, it must satisfy some other exit of ftype as well.

The type function generated for any total function is total. This is obvious since every recursive call of ftype in the definition decomposes X with at least one CAR or CDR (after checking that X is non-NIL).

As defined here, the output of _typeexpr_ is not normalated (although function bodies created by it are). For efficiency, a top-level version of the routine normalates the output of the procedure described here. Thus, the type expression of:

```
        (COND A (CONS B C) NIL)
```

is actually T, rather than:

```
    (COND (EQUAL X NIL) T (COND X (COND T T NIL) NIL)),
```

as would be generated by the basic routine. (The reader is invited to verify that the conditional above "optimizes" to T by normalation.)

Because boolean and numerically valued functions are common, _typeexpr_ actually checks for these explicitly with _boolean_ and _numeric_ (an inductive verifier for numerically valued expressions just like _boolean_). For such expressions, it specifically uses BOOLEAN and NUMBERP as the type functions. These two checks could be discarded at the expense of generating many different function names with definitions equivalent to BOOLEAN and NUMBERP (such as LENGTYPE derived earlier). It is important to realize that without these built-in checks, _typeexpr_ is still capable of recognizing these two types of expressions and generating precisely the right definitions.

Let us now consider some examples. From the previous discussion, it should be clear that _typeexpr_ generates the correct definition of LENGTYPE given (LENGTH A) (even if the boolean and numeric checks are not present).

Given (REVERSE A), _typeexpr_ (correctly) generates the function

```
(REVETYPE (LAMBDA (X) T)).
```

The function REVERSE ranges over the entire universe of lists. However, this is precisely correct only by coincidence. Upon encountering the call of APPEND in the definition of REVERSE, _typeexpr_ recursively determines that APPEND ranges over the entire universe (if its arguments are unrestricted). It then decides that REVERSE must also range over the entire universe -- without ever inspecting the particular restrictions upon the arguments in the call to APPEND in REVERSE. Proving that REVERSE ranges over the entire universe is a non-trivial theorem.

The function COUNT counts the number of occurrences of an element in a list:

```
(COUNT (LAMBDA (X Y)
       (COND Y
             (COND (EQUAL X (CAR Y))
                   (ADD1 (COUNT X (CDR Y)))
                   (COUNT X (CDR Y)))
             0))).
```

(Recall that the ADD1 term is normalated to (CONS NIL (COUNT X (CDR Y))).)

Given (COUNT A B), _typeexpr_ first finds the type expression of the definition above:

```
(COND (EQUAL X 0)
      T
      (COND (COUNTYPE X)
            T
            (COND X
                  (COND (EQUAL (CAR X) NIL)
                        (COUNTYPE (CDR X))
                        NIL)
                  NIL))).
```

Notice the recursive call of COUNTYPE which does not decompose X. This corresponds to the recursive call of COUNT which does not alter the output of the call before returning it. This means that the output of this recursive call was produced by some other exit of COUNT, and must therefore satisfy some other exit of COUNTYPE. Thus, (COUNTYPE X) is replaced, above, by NIL, and the expression is normalated to form the definition of COUNTYPE:

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

This is just NUMBERP again. As pointed out above, this type would be caught by the explicit numeric check in the actual typeexpr routine. The above description of its generation makes it clear that it is easy enough to write automatically.

Let PAIRLIST be defined by:

```
          (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))).
```

(PAIRLIST A B) generates an (association) list of pairs as long as A with the elements of A paired with those of B (or with NIL if B is short).

Given (PAIRLIST A B), typeexpr generates PAIRTYPE:

```
          (PAIRTYPE (LAMBDA (X)
                    (COND X
                          (COND (CAR X) (PAIRTYPE (CDR X)) NIL)
                          T))),
```

which correctly recognizes association lists.

As a final example, let BINTREE be a function which generates a full binary tree of NILs with a number representing the depth of the subtree at each node:

```
    (BINTREE (LAMBDA (X)
                 (COND X
                      (CONS (LENGTH X)
                           (CONS (BINTREE (CDR X)) (BINTREE (CDR X))))
                      NIL))).
```

Then for (BINTREE A), typeexpr generates BINTTYPE:

```
        (BINTTYPE (LAMBDA (X)
                      (COND X
                           (COND (LENGTYPE (CAR X))
                                (COND (CDR X)
                                     (COND (BINTTYPE (CAR (CDR X)))
                                          (BINTTYPE (CDR (CDR X)))
                                          NIL)
                                     NIL)
                                NIL)
                           T))).
```

This function recognizes only binary trees with a number at each node. The information that the tree is balanced, and the relationship between the number and the subtree at each node is lost. Note that in this example, typeexpr had to recursively write LENGTYPE for LENGTH before it could finish writing BINTTYPE.

We observe that typeexpr can generate reasonably efficient and often quite restrictive type functions. From the definition of typeexpr it is clear that an expression always satisfies its type expression. As in the case of BINTTREE, we see that the type expression generated may recognize other structures as well. A good (and devastating) example of this is obtained by finding the type expression for (SORT A). The result is:

```
        (SORTTYPE (LAMBDA (X) T)).
```

But SORT does not range over the entire universe. This is "almost true" however. Inspecting the definition, we see that NIL is a possible output, and so is anything output by ADDTOLIST (ignoring the arguments to the ADDTOLIST call in SORT). But one of the possible outputs of ADDTOLIST is (CONS X Y), where X and Y are the arguments. Thus, at first glance, the output of SORT can be anything.

Of course, the problem is that the call of ADDTOLIST in SORT restricts the arguments, so that while X can be anything, Y is a sorted list. Furthermore, (CONS X Y) is an output of ADDTOLIST only when X is LTE (less than or equal to) the CAR of Y. Since _typeexpr_ does not consider the arguments to non-primitive functions, and does not take advantage of the information available from the conditions tested, it misses this information.

As a result, in trying to prove:

```
(EQUAL (SORT (SORT A)) (SORT A)),
```

the program generalizes it to:

```
(EQUAL (SORT GENRL1) GENRL1),
```

where GENRL1 is completely unrestricted. While this is very reasonable, it is not a theorem. If _typeexpr_ could generate ORDERED as the (proper) type function for SORT, the theorem would correctly generalize to:

```
(IMPLIES (ORDERED GENRL1) (EQUAL (SORT GENRL1) GENRL1)),
```

which the theorem prover can prove.

The problem of generating ORDERED given SORT is still open, but appears to yield to a very similar approach, which takes into account the two kinds of information discussed above.

It is quite easy to take into account the types of the arguments to non-primitive functions. Before finding the type expression of a function application, one recursively finds the type expressions for each of the arguments. An association list is used to bind the variables in the function definition to their type expressions, and then the type expression for the definition is found.

Upon encountering a variable, its type expression is looked-up on the association list and returned (rather than assuming it could be bound to anything and returning T). Also, steps have to be taken to determine the type of a CAR or CDR expression, given the type of the argument. Several obvious heuristics work well.

Such a function has been implemented, but it is not used in the current theorem prover. This is because it produces more complicated function definitions and still does not solve the SORT problem mentioned above. However, the routine is capable of producing a type function which recognizes the type of:

```
(APPEND (LENGTH A) (LENGTH B))
```

to be a number.

Besides providing some protection against faulty generalizations, _typeexpr_ is interesting for two reasons. First, it very effectively uses the _eval_, _normalize_, and _reduce_ routines to optimize LISP code. This use of normalation was not originally forseen, but corresponds very strongly with the traditional way of writing programs: first write an inefficient but obvious algorithm and then use knowledge of the programming language to make it more efficient. The second reason _typeexpr_ is interesting is that it is an effective way to use automatic programming to aid the process of proving program correctness.

3.8 Induction

The induct routine is concerned with producing an induction formula which implies the theorem to be proved. This routine is appealed to only after normalation and fertilization have failed to rewrite the theorem and after generalization has replaced any common subterms as described. Two basic problems must be solved by induct. What term or terms should be inducted upon? What should be the exact form of the induction formula?

Both of these problems are solved by inspecting the fault descriptions generated by eval. Once the two problems are solved, induct creates the new expression which represents the induction formula, and this becomes the theorem to be proved.

The problem of choosing the terms to induct upon is discussed first. The routine that does this is a subroutine to induct, called pickindvars. The heuristic used, as explained in Chapter 2, is to induct upon the terms most likely to let the theorem "recurse back down" when those terms are made explicit CONSes. The hope is that the link between evaluation and induction will let a properly chosen induction conclusion evaluate into a properly chosen induction hypothesis. With the evaluation machinery at our disposal, this choice is usually easy.

If the theorem, p, is evaluated, it will not change, since normalation has already exhaustively rewritten the theorem. However, analysis will contain a set of fault descriptions, describing why the evaluation of the recursive expressions in p halted. The reader should recall the structure of analysis as presented in Section 3.2.

Briefly, each fault description corresponds to some recursive function application appearing in p. A description consists of a pair, composed of a bomb list and a failures list. The bomb list is a list of pockets indicating those terms simultaneously decomposed in a recursive call of the function. There is a pocket for each recursive call (which was halted) in the function's definition. The failures list is a list of all of the non-recursive CAR and CDR failures occurring in the evaluation of the definition.

pickindvars is concerned only with the terms in the bomb lists in _analysis_. The terms in the pockets are a series of CARs and CDRs applied to some term, which will be called the 'argument' (since it is an argument to the non-primitive function expression corresponding to the fault description). Call the set of (distinct) arguments occuring in the pockets of the bomb list of a fault description, the 'argument list' of that description.

Any argument in any argument list of _analysis_ might be considered as a term to induct upon (an 'induction candidate'). To a certain extent, if any such argument were replaced by an explicit CONS in the conclusion, the resulting expression would have a better chance of evaluating than it would have should some term not in an argument list be chosen. However, if an argument is not a skolem constant, it cannot be inducted upon without being generalized. Since it was not generalized, it cannot be considered as an induction candidate.

It is advisable to induct simultaneously upon all of the arguments associated with some fault description. If only a (proper) subset of the arguments were inducted upon, then only those arguments would be explicit CONSes in the conclusion. Since there would still be arguments in the fault description which were not inducted upon, the expression in the conclusion would again fail to recurse, because the remaining arguments in the description would again fail in recursion.

Therefore, any fault description with argument list containing a non-skolem constant is eliminated from consideration.

If any two argument lists share a term, then two recursive expressions depend upon that term being an explicit CONS. If the arguments in one of the lists are inducted upon (and thus made CONSes in the conclusion), then a CONS would be introduced into the expression corresponding to the fault description associated with the other argument list. This expression would not evaluate, because the other arguments in the argument list were not explicit CONSes. However, neither would it match the expression in the hypothesis from which it came, because of the "spurious" CONS introduced. Thus, the induction hypothesis would be prevented from matching (parts of) the evaluated conclusion. It is therefore a reasonable heuristic to merge any argument lists which share a skolem constant. (The bomb lists and failures lists associated with their fault descriptions are also merged. The result is still treated as a fault description with associated argument list, even though it now describes several faults.)

Often, after all possible merges, there is only one fault description left. If so, its associated argument list is chosen as the set of terms to be inducted upon (simultaneously). However, if there are multiple sets to choose from, the following heuristics are used.

Each list of arguments is "rated" by replacing, in p, each of the arguments in the list by an explicit CONS and then evaluating the result.

The number of rewrite rules successfully applied by eval is counted. This count gives an indication of how far evaluation was able to proceed under the particular list of arguments. Since induction on multiple terms simultaneously is bound to allow more (CAR and CDR) evaluation than induction on a single term, the count is divided by the number of arguments in the list. The list with the highest such rating is chosen as the set of induction terms.

For example, the theorem:

(EQUAL (APPEND A (APPEND B C)) (APPEND (APPEND A B) C)),

generates four fault descriptions. One of these is eliminated because it reports recursion on (APPEND A B), which cannot be inducted upon. Two of the others, reporting induction on A, are merged. This leaves two sets to choose from, one containing only A, the other only B. The above heuristic chooses A because it allows more evaluation. In particular, A allows the APPEND definition to be applied three times, while B allows its application only once.

Should several lists be tied for the highest rating, the one containing the greatest number of terms not previously inducted upon, in the current theorem, is chosen. This is merely to enable a new set of expressions to recurse.

Should this produce a tie, a random list is chosen from the "winners".

The terms in the argument list chosen are inducted upon simultaneously. The bomb and failures lists associated with this argument list are used by induct to set up the induction formula.

There must clearly be a NIL-case for each term inducted upon. The precise form of the induction step -- for example, whether there is a hypothesis about the CAR or the CDR or both -- is determined by inspection of the bomb and failures lists associated with the argument list of induction terms. A few examples of correct induction formulas and how they are generated using these lists may be helpful.

In proving the theorem:

```
(EQUAL (COPY A) A),
```

induct gets the above expression as its input (since normalation, fertilization, and generalization do not rewrite the theorem). Evaluation sets analysis to:

```
(((((CDR A))((CAR A))) NIL)),
```

which contains a single fault description, containing a bomb list with two pockets. pickindvars chooses A. Since only one term is being inducted upon, there need be only one NIL-case. Since the bomb list contains both (CAR A) and (CDR A), induction hypotheses about both components of the CONS supplied in the conclusion will be needed. Since A is never decomposed with more than one CAR or CDR (either recursively or non-recursively), a single explicit CONS in the conclusion is sufficient to guarantee that there will be no CAR or CDR failures in the evaluated definition. Thus, the induction hypothesis generated is:

```
(AND (EQUAL (COPY NIL) NIL)
     (IMPLIES (AND (EQUAL (COPY A1) A1)
                   (EQUAL (COPY A) A))
              (EQUAL (COPY (CONS A1 A)) (CONS A1 A)))).
```

The reader can verify that this formula normalates to T.

The next example is:

```
(IMPLIES (ELEMENT A B) (MEMBER (ELEMENT A B) B)),
```

where (ELEMENT A B) returns the Ath element of B (or NIL if B does not have A elements). This theorem normalates to:

```
(COND (ELEMENT A B) (MEMBER (ELEMENT A B) B) T),
```

and does not fertilize or generalize (because the hypothesis of an implicational conditional is protected). Call this expression p(A,B). After evaluation, analysis has three fault descriptions on it. They are:

```
((((CDR B))) ((CAR B))),
```

from the MEMBER expression; and two copies of:

```
((((CDR B)(CDR A))) ((CAR B))),
```

from the two (identical) ELEMENT expressions.

Since the three bomb lists are all linked by the common argument B, they are merged and pickindvars chooses simultaneous induction on A and B. This means that there must be two NIL-cases, one for each term inducted upon. The fact that only CDRs of the arguments are recursed upon (i.e., appear in the bomb list) implies a single induction hypothesis about them is sufficient. Again, no argument is decomposed by more than a single CAR or CDR, so a single explicit CONS in the conclusion is sufficient. The resulting formula is:

```
(AND (AND p(NIL,B) p(A,NIL))
     (IMPLIES p(A,B) p((CONS A1 A),(CONS B1 B)))).
```

Again, this formula normalates to T.

A final example is provided by the program's proof that the output of SORT is ORDERED. The original theorem is:

```
(ORDERED (SORT A)).
```

An initial induction on A, normalation, and generalization convert this to:

```
          (COND (ORDERED GENRL1) (ORDERED (ADDTOLIST A1 GENRL1)) T).
```

Call this p(GENRL1,A1).

Evaluation of this expression sets analysis to a list containing three fault descriptions; however, one of them is eliminated because it would require induction on (ADDTOLIST A1 GENRL1). The remaining two are:

```
          ((((CDR GENRL1))) ((CAR GENRL1)(CAR GENRL1))),
```

from the ADDTOLIST term, and:

```
          ((((CDR GENRL1))) ((CAR (CDR GENRL1))(CDR GENRL1)
                             (CAR GENRL1)(CDR GENRL1))),
```

from the ORDERED term in the hypothesis of the theorem.

<u>pickindvars</u> chooses GENRL1, so there is one NIL-case. The fact that recursion occurs only on the CDR of GENRL1 indicates that an induction hypothesis about only the CDR of the supplied CONS in the conclusion will be sufficient. However, the presence of the term (CAR (CDR GENRL1)) on the failures list indicates that there should be two explicit CONSes in the conclusion to allow it to evaluate without (non-recursive) failure. This means the hypothesis will be about the inner CONS of the conclusion, and that, for soundness, a special case (the general list of length 1) must be considered. The output of induct is:

```
          (AND p(NIL,A1)
               (AND p((CONS GENRL11 NIL),A1)
                    (IMPLIES p((CONS GENRL12 GENRL1),A1)
                             p((CONS GENRL11 (CONS GENRL12 GENRL1)),A1)))).
```

This is just the representation for:

```
p(NIL,A1) & ∀X(p((CONS X NIL),A1) &
                 ∀Y,Z(p((CONS Y Z),A1) -> p((CONS X (CONS Y Z)),A1)).
```

This is called 'special induction' because of the provision of a second basis (the case for the list of length 1).

Although far from obvious, the induction formula above **normalates** to:

```
(COND (LTE A1 GENRL11) T (LTE GENRL11 A1)),
```

which is the theorem that either A1 is less than or equal to GENRL11, or vice versa. This is proved by simultaneous induction on A1 and GENRL11, which can be very easily verified. The program's output for the (ORDERED (SORT A)) theorem is given in Chapter 4.

This sample of induction formulas illustrates some of the basic induction schemes the _induct_ routine can generate. It can mix two schemes when necessary. For example, it can do simultaneous induction on two variables with induction hypotheses about the CARs and CDRs of both. It can also do induction on n variables, or generate required but unusual combinations of induction hypotheses. With the exception of special induction, the induction formula is generated by a general mechanism which maps the structure of the bomb list into the required formula. While this mechanism could be generalized for some cases of special induction, the occasions requiring it are so rare that it is handled separately. _induct_ will now be described.

Given a theorem, _induct_ first uses _pickindvars_ to select a list of skolem constants to be inducted upon simultaneously. For each term in that list it generates a NIL-case by replacing the term by NIL in the theorem. These NIL-cases are then conjoined with ANDs.

If the bomb or failures lists contain a "deep" term, such as:

(CAR (CDR A)),

where A is being inducted upon, this indicates that more than one CONS is required in the conclusion, and that a second basis is needed. As a result, the program enters a special mode, described below.

If only one CONS per term is sufficient, the program generates the skolem constants to be used as the CARs of the CONSes replacing each induction term in the conclusion. These are called the 'CAR-constants' of their respective induction terms. (The analogous 'CDR-constants' are the induction terms themselves. This merely reduces the number of new symbols the user is confronted with.) If arg_1, ... , arg_n are the terms being inducted upon, let $cararg_1$, ... , $cararg_n$, be their respective CAR-constants.

In general the induction hypothesis is a conjunction of hypotheses. These involve the CARs and CDRs of the CONSes which are introduced into the conclusion. The precise combinations of CARs and CDRs used are dictated by the pockets on the bomb list.

If A is being inducted upon, and (CAR A) (or (CDR A)) does not occur on the bomb list, then no hypothesis need involve the CAR-constant (or, the CDR-constant) for A. If both (CAR A) and (CDR A) occur, then the hypothesis should be the conjunction of one for the CAR-constant and one for the CDR-constant.

The above guidelines completely specify the hypothesis generated for the case of induction on a single term. The hypothesis is either $p(arg_1)$, or $p(cararg_1)$, or (AND $p(cararg_1)$ $p(arg_1)$), according to whether (CDR arg_1), (CAR arg_1), or both occur on the bomb (where $p(arg_1)$ is the original theorem input to induct).

The case for n term induction is much more complicated, and is not handled in its full generality by the program. However, the behaviour of the program is quite acceptable on theorems about the functions in Appendix A.

If more than one term is being inducted upon, induct merges all those pockets on the bomb list which contain only CDR failures. It also merges all those which contain only CAR failures. This produces two 'super-pockets'. It is usually the case that no other pockets are left. Such pockets would be mixed CARs and CDRs, and very few functions recurse on the CAR of one argument, while recursing (in the same call) on the CDR of another. Certainly no function in Appendix A, which includes many common list processing functions, exhibits this kind of recursion.

For this reason, the program uses these two super-pockets to determine the hypothesis. Two hypotheses will be generated and then conjoined. The first is generated from the CAR super-pocket, by replacing every argument in that pocket by its CAR-constant, in the theorem, p. Any term inducted upon not occurring in the CAR super-pocket is left alone in this hypothesis. Such a term must occur in the CDR super-pocket, and since the CDR-constant of a term is just itself, the hypothesis generated for the CAR super-pocket concerns the CAR-constants of those terms in that super-pocket, and the CDR-constants of those terms not in that super-pocket.

An analogous procedure is used to form a hypothesis from the CDR super-pocket. Any argument occurring in it is left as it is; any induction term not occurring in it (and thus, occurring in the CAR super-pocket) is replaced by its CAR-constant.

These two hypotheses are conjoined with AND and constitute the induction hypothesis generated by induct.

Finally, the conclusion is formed by substituting (CONS cararg$_i$ arg$_i$) for arg$_i$ in p.

The hypothesis and conclusion are then put into an IMPLIES expression, and then this is conjoined with the NIL-case(s) with AND. The resulting expression is returned as the new theorem to be proved.

The need for special induction arises so seldomly that it was not felt necessary to write the general routine for generating such induction formulas, for say, single term induction. As a result, only two special induction forms can be generated. Both of these allow induction on a single term only. As usual, the configuration of the bomb list and failures list suggests which form to use.

The form exhibited in the third example above is used for induction on a single term, when the bomb list contains only (CDR arg), but non-recursive failures of the form (CAR (CDR arg)) or (CDR (CDR arg)) occurred, where arg is the term being inducted upon. The induction formula generated is:

```
(AND p(NIL)
     (AND p((CONS cararg1 NIL))
          (IMPLIES p((CONS cararg2 arg))
                   p((CONS cararg1 (CONS cararg2 arg)))))),
```

where cararg$_1$ and cararg$_2$ are new skolem constants, and p(arg) was the input to induct. The relationship between such a formula and the fault description is clear.

A second special form is generated for functions which CDR their arguments twice in recursion. See the definitions of HALF, EVEN2, or SWAPTREE for example. This condition is recognized by the occurrence on the bomb list of terms such as (CAR (CDR arg)) or (CDR (CDR arg)). The induction formula generated is:

```
(AND p(NIL)
     (AND p((CONS cararg₁ NIL))
          (IMPLIES hyp p((CONS cararg₁ (CONS cararg₂ arg)))))),
```

where the hypothesis, hyp, is a conjunction (in general) of at most three expressions:

p(cararg_1), if (CAR arg) occurs on the bomb list,

p(cararg_2), if (CAR (CDR arg)) occurs on the bomb list,

and

p(arg), if (CDR (CDR arg)) occurs on the bomb list.

Again, the relationship between this formula and the structure of the bomb list should be clear. For an example of this type of induction, see the program's proof of:

(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A))),

in Chapter 4.

It should be emphasized that the program does not recognize theorems or function names in order to generate these induction formulas. It recognizes configurations of the bomb and failures lists. Furthermore, it should be clear that the generation can be generalized for single inductions involving deeper nestings of CARs or CDRs. As remarked, such a routine is not justified by its usefulness.

These two special forms handle a wide variety of cases. The first is used when functions involved in the theorem recurse "one step" at a time, but inspect adjacent elements of their recursive argument. The second is used for functions which recurse down their argument "two steps" per call.

Only 10% of the theorems listed in Appendix B require special induction formulas. The remaining 90% use formulas generated by the more general induction mechanism.

3.9 Technical Information

The program is written in the programming language POP-2, developed by Rod Burstall and Robin Popplestone, of the Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh. It runs on the ICL 4130 belonging to the School.

The compiled code occupies about 15K of 24-bit words. The various routines are written in a fairly elegant POP-2 style. Except for assignment to property lists, there is no destructive assignment, and side effects are avoided (the main exception is analysis). The program has a highly modular, structured design, and the more natural quick and dirty techniques were consciously avoided to keep the program easily modifiable. (See Dahl, Dijkstra, and Hoare 1972, for a discussion of this type of programming.)

The majority of theorem prover's time (well over 75%) is spent in normalation. Since these routines are composed entirely of recursive, constructive list processing, and apply the rewrite rules in the straightforward manner described, they effectively copy large formulas many times during the course of normalation. Since an expression is completely normalized before it is reduced, this process often creates quite large intermediate expressions.

Despite these inefficiencies, the "typical" theorem proved requires only 8 to 10 seconds of CPU time. For comparison purposes, it should be noted that the time for cons in 4130 POP-2 is 400 microseconds, and _car_ and _cdr_ are about 50 microseconds each. The "hardest" theorems proved, such as those involving SORT, require 40 to 150 seconds each.

The design of the program, especially the straightforward approach of "hitting" the theorem over and over again with rewrite rules until it can no longer be changed, is largely due to the influence of Woody Bledsoe.

CHAPTER 4 DETAILED EXAMPLES OF PROGRAM OUTPUT

4.1 Introduction to the Examples

The following four Sections present sample output from the theorem prover. The theorems proved are:

```
(IMPLIES (OR (MEMBER A B) (MEMBER A C)) (MEMBER A (APPEND B C))),
(EQUAL (REVERSE (REVERSE A)) A),
(ORDERED (SORT A)),
```

and

```
(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A))).
```

Complete proofs are given for the first three theorems; the proof of the fourth is terminated after a familiar lemma is produced as the theorem to be proved.

The formula representing the theorem is pretty-printed each time it is transformed by one of the basic routines. A heading above the formula explains which routine was responsible for the transformation. The pretty-print routine is explained in Boyer 1973 . Dots are printed within the formulas to help the reader follow the indentations. Printing of formulas requiring more than 60 lines has been surpressed in these examples. The message (TOO BIG) is printed in the place of such a formula.

The reader is referred to Appendix A for the definitions of the LISP functions involved in these examples.

Notes have been added to the proof by the author. These are typed in lower-case and point out interesting lemmas generated during the proof or explain certain steps. Except for these notes, all of the following is produced automatically by the theorem prover.

4.2 Sample Output 1

This theorem states an obvious relationship between MEMBER and APPEND. The proof illustrates how evaluation eliminates non-recursive functions (such as IMPLIES and OR), and how it generates a case-analysis which normalates to T.

```
THEOREM TO BE PROVED:

(IMPLIES (OR (MEMBER A B) (MEMBER A C)) (MEMBER A (APPEND B C)))

EVALUATION YIELDS:

(COND (COND (MEMBER A B) T (COND (MEMBER A C) T NIL))
      (COND (MEMBER A (APPEND B C)) T NIL)
      T)

WHICH NORMALIZES TO:

(COND (MEMBER A B)
      (MEMBER A (APPEND B C))
      (COND (MEMBER A C) (MEMBER A (APPEND B C)) T))



MUST TRY INDUCTION.

INDUCT ON B.

THE THEOREM TO BE PROVED IS NOW:

(AND
 (COND (MEMBER A NIL)
 .      (MEMBER A (APPEND NIL C))
 .      (COND (MEMBER A C) (MEMBER A (APPEND NIL C)) T))
 (IMPLIES
    (COND (MEMBER A B)
    .      (MEMBER A (APPEND B C))
    .      (COND (MEMBER A C) (MEMBER A (APPEND B C)) T))
    (COND (MEMBER A (CONS B1 B))
           (MEMBER A (APPEND (CONS B1 B) C))
           (COND (MEMBER A C) (MEMBER A (APPEND (CONS B1 B) C)) T))))

EVALUATION YIELDS:

(COND
 (COND (MEMBER A C) (MEMBER A C) T)
 (COND
 .(COND
 .   (COND (MEMBER A B)
 .   .      (MEMBER A (APPEND B C))
 .   .     (COND (MEMBER A C) (MEMBER A (APPEND B C)) T))
 .   (COND (COND (COND (EQUAL A B1) T (MEMBER A B))
 .   .               (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .   .               (COND (MEMBER A C)
 .   .                     (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .   .                     T))
 .   .     T
 .   .     NIL)
 .   T)
 .T
 .NIL)
 NIL)
```

WHICH NORMALIZES TO:

```
(COND
 (COND (MEMBER A C) (MEMBER A C) T)
 (COND
 .(MEMBER A B)
 .(COND
 .   (MEMBER A (APPEND B C))
 .   (COND (EQUAL A B1)
 .   .     (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .   .     (COND (MEMBER A B)
 .   .           (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .   .           (COND (MEMBER A C)
 .   .                 (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .   .                 T)))
 .   T)
 .(COND
 . (MEMBER A C)
 . (COND
 . . (MEMBER A (APPEND B C))
 . . (COND (EQUAL A B1)
 . . .     (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 . . .     (COND (MEMBER A B)
 . . .           (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 . . .           (COND (MEMBER A C)
 . . .                 (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 . . .                 T)))
 . . T)
 . (COND (EQUAL A B1)
 .       (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .       (COND (MEMBER A B)
 .             (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .             (COND (MEMBER A C)
 .                   (COND (EQUAL A B1) T (MEMBER A (APPEND B C)))
 .                   T)))))
 NIL)
```

AND REDUCES TO:

```
(COND
 (COND (MEMBER A C) T T)
 (COND
 .    (MEMBER A B)
 .    (COND (MEMBER A (APPEND B C)) (COND (EQUAL A B1) T T) T)
 .    (COND (MEMBER A C)
 .          (COND (MEMBER A (APPEND B C)) (COND (EQUAL A B1) T T) T)
 .          (COND (EQUAL A B1) T T)))
 NIL)

WHICH NORMALIZES TO:

T
```

4.3 Sample Output 2

This theorem states that REVERSE is idempotent. The proof involves fertilization and generalization.

```
THEOREM TO BE PROVED:

(EQUAL (REVERSE (REVERSE A)) A)



MUST TRY INDUCTION.

INDUCT ON A.

THE THEOREM TO BE PROVED IS NOW:

(AND (EQUAL (REVERSE (REVERSE NIL)) NIL)
     (IMPLIES (EQUAL (REVERSE (REVERSE A)) A)
              (EQUAL (REVERSE (REVERSE (CONS A1 A))) (CONS A1 A))))

EVALUATION YIELDS:

(COND
 (COND
 .(EQUAL (REVERSE (REVERSE A)) A)
 .(COND
 .  (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL))) (CONS A1 A))
 .  T
 .  NIL)
 .T)
 T
 NIL)

WHICH NORMALIZES TO:

(COND
    (EQUAL (REVERSE (REVERSE A)) A)
    (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL))) (CONS A1 A))
    T)

FERTILIZE WITH (EQUAL (REVERSE (REVERSE A)) A).

THE THEOREM TO BE PROVED IS NOW:

(COND (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL)))
             (CONS A1 (REVERSE (REVERSE A))))
      T
      (* 1))

GENERALIZE COMMON SUBTERMS BY REPLACING (REVERSE A) BY GENRL1.

THE GENERALIZED TERM IS:
```

```
(COND (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
             (CONS A1 (REVERSE GENRL1)))
      T
      (* 1))
```

Note: After fertilizing with the induction hypothesis and generalizing, a lemma must be proved. This lemma links APPEND and REVERSE in an interesting way. When this link has been proved, the original theorem will have been proved. Induction on GENRL1 is used.

```
MUST TRY INDUCTION.

INDUCT ON GENRL1.

THE THEOREM TO BE PROVED IS NOW:

(AND
 (COND (EQUAL (REVERSE (APPEND NIL (CONS A1 NIL)))
 .             (CONS A1 (REVERSE NIL)))
 .      T
 .      (* 1))
 (IMPLIES
  (COND (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
  .             (CONS A1 (REVERSE GENRL1)))
  .      T
  .      (* 1))
  (COND
       (EQUAL (REVERSE (APPEND (CONS GENRL11 GENRL1) (CONS A1 NIL)))
       .       (CONS A1 (REVERSE (CONS GENRL11 GENRL1))))
       T
       (* 1))))

EVALUATION YIELDS:

(COND
 (COND
 .(COND (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .             (CONS A1 (REVERSE GENRL1)))
 .      T
 .      (* 1))
 .(COND
 . (COND
 . .  (EQUAL (APPEND (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 . .  .               (CONS GENRL11 NIL))
 . .  .       (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
 . .  T
 . .  (* 1))
 . T
 . NIL)
 .T)
 T
 NIL)
```

WHICH NORMALIZES TO:

```
(COND
 (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .       (CONS A1 (REVERSE GENRL1)))
 (COND
 .    (EQUAL (APPEND (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .    .              (CONS GENRL11 NIL))
 .    .       (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
 .    T
 .    (* 1))
 (COND
  (* 1)
  (COND
  .   (EQUAL (APPEND (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
  .   .               (CONS GENRL11 NIL))
  .   .       (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
  .   T
  .   (* 1))
  T))
```

AND REDUCES TO:

```
(COND
 (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .       (CONS A1 (REVERSE GENRL1)))
 (COND
 .    (EQUAL (APPEND (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .    .              (CONS GENRL11 NIL))
 .    .       (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
 .    T
 .    (* 1))
 (COND
  (* 1)
  (COND
  .   (EQUAL (APPEND (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
  .   .               (CONS GENRL11 NIL))
  .   .       (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
  .   T
  .   T)
  T))
```

WHICH NORMALIZES TO:

```
(COND
 (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .       (CONS A1 (REVERSE GENRL1)))
 (COND
 .    (EQUAL (APPEND (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
 .    .              (CONS GENRL11 NIL))
 .    .       (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
 .    T
 .    (* 1))
 T)
```

```
FERTILIZE WITH (EQUAL (REVERSE (APPEND GENRL1 (CONS A1 NIL)))
                      (CONS A1 (REVERSE GENRL1))).

THE THEOREM TO BE PROVED IS NOW:

(COND
 (COND
 .    (EQUAL (APPEND (CONS A1 (REVERSE GENRL1)) (CONS GENRL11 NIL))
 .    .      (CONS A1 (APPEND (REVERSE GENRL1) (CONS GENRL11 NIL))))
 .    T
 .    (* 1))
 T
 (* 2))

EVALUATION YIELDS:

T
```

4.4 Sample Output 3

Below is the program's proof that the output of SORT is ORDERED.

```
THEOREM TO BE PROVED:

(ORDERED (SORT A))



MUST TRY INDUCTION.

INDUCT ON A.

THE THEOREM TO BE PROVED IS NOW:

(AND (ORDERED (SORT NIL))
     (IMPLIES (ORDERED (SORT A)) (ORDERED (SORT (CONS A1 A)))))

EVALUATION YIELDS:

(COND (COND (ORDERED (SORT A))
            (COND (ORDERED (ADDTOLIS A1 (SORT A))) T NIL)
            T)
      T
      NIL)

WHICH NORMALIZES TO:

(COND (ORDERED (SORT A)) (ORDERED (ADDTOLIS A1 (SORT A))) T)

GENERALIZE COMMON SUBTERMS BY REPLACING (SORT A) BY GENRL1.

THE GENERALIZED TERM IS:

(COND (ORDERED GENRL1) (ORDERED (ADDTOLIS A1 GENRL1)) T)
```

Note: This generalization captures one of the essential properties of ADDTOLIS: its output is ORDERED if its second argument is ORDERED. This non-trivial lemma has been produced entirely automatically. It is proved by induction on GENRL1. Because of the way ORDERED decomposes its argument, an unusual induction formula is generated. The induction hypothesis is that the theorem holds for (CONS GENRL12 GENRL1), and the conclusion is that it holds for (CONS GENRL11 (CONS GENRL12 GENRL1)). A second basis is also required.

```
MUST TRY INDUCTION.

(SPECIAL CASE REQUIRED)

INDUCT ON GENRL1.

THE THEOREM TO BE PROVED IS NOW:
```

```
(AND
 (COND (ORDERED NIL) (ORDERED (ADDTOLIS A1 NIL)) T)
 (AND
  (COND (ORDERED (CONS GENRL11 NIL))
  .     (ORDERED (ADDTOLIS A1 (CONS GENRL11 NIL)))
  .     T)
  (IMPLIES
   (COND (ORDERED (CONS GENRL12 GENRL1))
   .     (ORDERED (ADDTOLIS A1 (CONS GENRL12 GENRL1)))
   .     T)
   (COND
        (ORDERED (CONS GENRL11 (CONS GENRL12 GENRL1)))
        (ORDERED (ADDTOLIS A1 (CONS GENRL11 (CONS GENRL12 GENRL1))))
        T))))
```

EVALUATION YIELDS:

```
(COND
 (COND
 .(ORDERED (COND (LTE A1 GENRL11)
 .               (CONS A1 (CONS GENRL11 NIL))
 .               (CONS GENRL11 (CONS A1 NIL))))
 .(COND
 . (COND
 . .(COND
 . .    (COND GENRL1
 . .    .     (COND (LTE GENRL12 (CAR GENRL1)) (ORDERED GENRL1) NIL)
 . .    .     T)
 . .    (ORDERED (COND (LTE A1 GENRL12)
 . .    .              (CONS A1 (CONS GENRL12 GENRL1))
 . .    .              (CONS GENRL12 (ADDTOLIS A1 GENRL1))))
 . .    T)
 . .(COND
 . . (COND
 . . .(COND
 . . .  (LTE GENRL11 GENRL12)
 . . .  (COND GENRL1
 . . . .      (COND (LTE GENRL12 (CAR GENRL1)) (ORDERED GENRL1) NIL)
 . . . .      T)
 . . .  NIL)
 . . .(ORDERED
 . . .     (COND (LTE A1 GENRL11)
 . . .           (CONS A1 (CONS GENRL11 (CONS GENRL12 GENRL1)))
 . . .           (CONS GENRL11
 . . .                 (COND (LTE A1 GENRL12)
 . . .                       (CONS A1 (CONS GENRL12 GENRL1))
 . . .                       (CONS GENRL12 (ADDTOLIS A1 GENRL1))))))
 . . .T)
 . . T
 . . NIL)
 . .T)
 . T
 . NIL)
 .NIL)
 T
 NIL)

WHICH NORMALIZES TO:

(TOO BIG)

AND REDUCES TO:

(TOO BIG)

EVALUATION YIELDS:

(TOO BIG)

WHICH NORMALIZES TO:

(TOO BIG)

AND REDUCES TO:

(COND
 (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
 (COND
 .GENRL1
 .(COND
 . (LTE GENRL12 (CAR GENRL1))
 . (COND
 . .(ORDERED GENRL1)
 . .(COND
 . . (LTE A1 GENRL12)
 . . (COND (LTE GENRL11 GENRL12)
 . . .     (COND (LTE A1 GENRL11) T (COND (LTE GENRL11 A1) T NIL))
 . . .     T)
 . . (COND
 . .  (ADDTOLIS A1 GENRL1)
 . .  (COND
 . .  .(LTE GENRL12 (CAR (ADDTOLIS A1 GENRL1)))
 . .  .(COND
 . . .    (ORDERED (ADDTOLIS A1 GENRL1))
 . . .    (COND (LTE GENRL11 GENRL12) (COND (LTE A1 GENRL11) T T) T)
 . . .    T)
 . .  .T)
 . .  (COND (LTE GENRL11 GENRL12) (COND (LTE A1 GENRL11) T T) T)))
 . .T)
 . T)
 .(COND
 .  (LTE A1 GENRL12)
 .  (COND (LTE GENRL11 GENRL12)
 . .      (COND (LTE A1 GENRL11) T (COND (LTE GENRL11 A1) T NIL))
 . .      T)
 .  (COND (LTE GENRL12 A1)
 .        (COND (LTE GENRL11 GENRL12) (COND (LTE A1 GENRL11) T T) T)
 .        T)))
 NIL)
```

```
WHICH NORMALIZES TO:

(COND
 (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
 (COND
 .GENRL1
 .(COND (LTE GENRL12 (CAR GENRL1))
 .      (COND (ORDERED GENRL1)
 .            (COND (LTE A1 GENRL12)
 .                  (COND (LTE GENRL11 GENRL12)
 .                        (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
 .                        T)
 .                  T)
 .            T)
 .      T)
 .(COND (LTE A1 GENRL12)
 .      (COND (LTE GENRL11 GENRL12)
 .            (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
 .            T)
 .      T))
 NIL)

AND REDUCES TO:

(COND
 (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
 (COND
 .GENRL1
 .(COND
 .  (LTE GENRL12 (CAR GENRL1))
 .  (COND (ORDERED GENRL1)
 .  .     (COND (LTE A1 GENRL12) (COND (LTE GENRL11 GENRL12) T T) T)
 .  .     T)
 .  T)
 .(COND (LTE A1 GENRL12) (COND (LTE GENRL11 GENRL12) T T) T))
 NIL)

WHICH NORMALIZES TO:

(COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
```

Note: The hideous intermediate expressions above have been normalated to the statement that A1 is less than or equal to GENRL11 or vice versa. Although this is an elementary fact about LTE, it was generated as a necessary lemma automatically, and it must be proved by induction. Because LTE recurses on both arguments simultaneously, induction on both A1 and GENRL11 is used. The proof is immediate and illustrates the value of induction on n variables in cases such as this.

```
MUST TRY INDUCTION.
```

```
INDUCT ON GENRL11 AND A1.

THE THEOREM TO BE PROVED IS NOW:

(AND (AND (COND (LTE A1 NIL) T (LTE NIL A1))
          (COND (LTE NIL GENRL11) T (LTE GENRL11 NIL)))
     (IMPLIES (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
              (COND (LTE (CONS A11 A1) (CONS GENRL111 GENRL11))
                    T
                    (LTE (CONS GENRL111 GENRL11) (CONS A11 A1)))))

EVALUATION YIELDS:

(COND
 (COND (COND (COND A1 NIL T) T T) T NIL)
 (COND (COND (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
 .           (COND (COND (LTE A1 GENRL11) T (LTE GENRL11 A1)) T NIL)
 .           T)
 .     T
 .     NIL)
 NIL)

WHICH NORMALIZES TO:

(COND (LTE A1 GENRL11)
      (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
      (COND (LTE GENRL11 A1)
            (COND (LTE A1 GENRL11) T (LTE GENRL11 A1))
            T))

AND REDUCES TO:

(COND (LTE A1 GENRL11) T (COND (LTE GENRL11 A1) T T))

WHICH NORMALIZES TO:

T
```

Thus, the theorem that SORT produces an ORDERED list has been proved. It is important to note that no auxilary information about ADDTOLIS or LTE was used except for those lemmas generated automatically. The lemmas required were that ADDTOLIS produces an ORDERED list when its second argument is ORDERED, and that for all X and Y, either X is less than or equal to Y, or vice versa. These lemmas were proved by induction, using only the definitions of the functions involved.

In order to establish the correctness of SORT, the theorem that its output is a permutation of its input must be proved. The program can prove this theorem as well. Its proof is not exhibited here however.

4.5 Sample Output 4

FLATTEN constructs a list of the tips of a binary tree. Nodes in the tree are recognized by NODE and constructed by CONSNODE. SWAPTREE interchanges the two branches of every node in a tree. The theorem states a relationship between FLATTEN, SWAPTREE, and REVERSE. The proof illustrates an induction formula for tree-structured terms.

```
THEOREM TO BE PROVED:

(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))


MUST TRY INDUCTION.

(SPECIAL CASE REQUIRED)

INDUCT ON A.

THE THEOREM TO BE PROVED IS NOW:

(AND
 (EQUAL (FLATTEN (SWAPTREE NIL)) (REVERSE (FLATTEN NIL)))
 (AND
  (EQUAL (FLATTEN (SWAPTREE (CONS A1 NIL)))
  .      (REVERSE (FLATTEN (CONS A1 NIL))))
  (IMPLIES
         (AND (EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
         .    (EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A))))
         (EQUAL (FLATTEN (SWAPTREE (CONS A1 (CONS A2 A))))
                (REVERSE (FLATTEN (CONS A1 (CONS A2 A))))))))
```

Note: The induction hypothesis is a conjunction. The first conjunct states that the theorem holds for A2. The second states that it holds for A. The conclusion is that it holds for (CONS A1 (CONS A2 A)). A second basis is also required. This formula is generated because FLATTEN recurses on both the CAR of the CDR and the CDR of the CDR of its argument.

```
EVALUATION YIELDS:

(COND
 (COND
 .(EQUAL (FLATTEN (COND (COND A1 NIL NIL)
 .                      (CONS NIL (CONS NIL NIL))
 .                      (CONS A1 NIL)))
 .       (REVERSE (COND (COND A1 NIL NIL)
 .                      (CONS NIL (CONS NIL NIL))
 .                      (CONS (CONS A1 NIL) NIL))))
 .(COND
 . (COND
 . .(COND (EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
 . .     (COND (EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 . .           T
 . .           NIL)
 . .     NIL)
 . .(COND
 . . (EQUAL
 . . .   (FLATTEN (COND (COND A1 NIL T)
 . . .   .              (CONS NIL (CONS (SWAPTREE A) (SWAPTREE A2)))
 . . .   .              (CONS A1 (CONS A2 A))))
 . . .   (REVERSE (COND (COND A1 NIL T)
 . . .                  (APPEND (FLATTEN A2) (FLATTEN A))
 . . .                  (CONS (CONS A1 (CONS A2 A)) NIL))))
 . . T
 . . NIL)
 . .T)
 . T
 . NIL)
 .NIL)
 T
 NIL)
```

WHICH NORMALIZES TO:

```
(COND
 (EQUAL (FLATTEN (CONS A1 NIL)) (REVERSE (CONS (CONS A1 NIL) NIL)))
 (COND
 .(EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
 . (COND
 . (EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 . (COND
 . .A1
 . .(COND A1
 . .     (EQUAL (FLATTEN (CONS A1 (CONS A2 A)))
 . .            (REVERSE (CONS (CONS A1 (CONS A2 A)) NIL)))
 . .     (EQUAL (FLATTEN (CONS A1 (CONS A2 A)))
 . .            (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 . .(COND
 . .   A1
 . .   (EQUAL (FLATTEN (CONS NIL (CONS (SWAPTREE A) (SWAPTREE A2))))
 . .   .      (REVERSE (CONS (CONS A1 (CONS A2 A)) NIL)))
 . .   (EQUAL (FLATTEN (CONS NIL (CONS (SWAPTREE A) (SWAPTREE A2))))
 . .          (REVERSE (APPEND (FLATTEN A2) (FLATTEN A))))))
 . T)
 .T)
 NIL)
```

AND REDUCES TO:

```
(COND
 (EQUAL (FLATTEN (CONS A1 NIL)) (REVERSE (CONS (CONS A1 NIL) NIL)))
 (COND
 .(EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
 .(COND
 . (EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 . (COND
 . .   A1
 . .   (EQUAL (FLATTEN (CONS A1 (CONS A2 A)))
 . .   .       (REVERSE (CONS (CONS A1 (CONS A2 A)) NIL)))
 . .   (EQUAL (FLATTEN (CONS NIL (CONS (SWAPTREE A) (SWAPTREE A2))))
 . .           (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 . T)
 .T)
 NIL)
```

EVALUATION YIELDS:

```
(COND
 (EQUAL (COND (COND A1 NIL NIL)
 .             (CONS NIL (CONS NIL NIL))
 .             (CONS (CONS A1 NIL) NIL))
 .       (CONS (CONS A1 NIL) NIL))
 (COND
 .(EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
 .(COND
 . (EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 . (COND
 . .  A1
 . .  (EQUAL (COND (COND A1 NIL T)
 . . .             (APPEND (FLATTEN A2) (FLATTEN A))
 . . .             (CONS (CONS A1 (CONS A2 A)) NIL))
 . . .       (CONS (CONS A1 (CONS A2 A)) NIL))
 . .  (EQUAL (APPEND (FLATTEN (SWAPTREE A)) (FLATTEN (SWAPTREE A2)))
 . .          (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 . T)
 .T)
 NIL)
```

WHICH NORMALIZES TO:

```
(COND
 (EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
 (COND
 .(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 .(COND
 .    A1
 .    (COND A1
 .    .     T
 .    .     (EQUAL (APPEND (FLATTEN A2) (FLATTEN A))
 .    .            (CONS (CONS A1 (CONS A2 A)) NIL)))
 .    (EQUAL (APPEND (FLATTEN (SWAPTREE A)) (FLATTEN (SWAPTREE A2)))
 .           (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 .T)
 T)
```

AND REDUCES TO:

```
(COND
 (EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2)))
 (COND
 .(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 .(COND
 .    A1
 .    T
 .    (EQUAL (APPEND (FLATTEN (SWAPTREE A)) (FLATTEN (SWAPTREE A2)))
 .           (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 .T)
 T)
```

FERTILIZE WITH (EQUAL (FLATTEN (SWAPTREE A2)) (REVERSE (FLATTEN A2))).

THE THEOREM TO BE PROVED IS NOW:

```
(COND
 (COND
 .(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))
 .(COND
 .     A1
 .     T
 .     (EQUAL (APPEND (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A2)))
 .            (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 .T)
 T
 (* 1))
```

FERTILIZE WITH (EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A))).

THE THEOREM TO BE PROVED IS NOW:

```
(COND
 (COND
 .(COND A1
 .      T
 .      (EQUAL (APPEND (REVERSE (FLATTEN A)) (REVERSE (FLATTEN A2)))
 .             (REVERSE (APPEND (FLATTEN A2) (FLATTEN A)))))
 .T
 .(* 2))
 T
 (* 1))
```

GENERALIZE COMMON SUBTERMS BY REPLACING (FLATTEN A) BY GENRL1 AND (FLATTEN A2) BY GENRL2.

THE GENERALIZED TERM IS:

```
(COND (COND (COND A1
                  T
                  (EQUAL (APPEND (REVERSE GENRL1) (REVERSE GENRL2))
                         (REVERSE (APPEND GENRL2 GENRL1))))
            T
            (* 2))
      T
      (* 1))
```

The above formula has been produced by fertilizing with both conjuncts of the induction hypothesis, and then generalizing. The formula represents the familiar theorem that REVERSE can be distributed over APPEND (if the arguments to the APPEND are swapped). This theorem was proved in Section 2.3. The proof generates two nice lemmas: NIL is a right-identity for APPEND; and APPEND is associative. Since the proof has been described in Section 2.3, it is omitted here.

CHAPTER 5 EXTENSIONS

5.1 Termination

Up to now we have not discussed the notion of termination. This section is devoted to two topics concerned with termination. The first is characterizing the behaviour of the present program, formally and practically, in the presence of partial functions. The second topic is the value of the present approach to automatically proving program termination.

Formally, to discuss the notion of termination, we must define how the value of an expression is calculated. We have not done this in the theory and will not do so here. Instead, we will rely upon the reader's intuitions about LISP when discussing this question.

Since the theorem prover is sound, any instance of a formula which is proved by it is equivalent to T, provided the theory in which the theorem is proved is consistent. If a traditional interpreter (such as our _eval_ modified so that it is no longer a symbolic interpreter) is used to decide whether the instance is T, it may not always terminate. However, any instance which does terminate when evaluated, yields T.

For example, consider the function defined by:

```
(CHOP (LAMBDA (X Y)
      (COND (EQUAL X (CAR Y))
            (CDR Y)
            (CHOP X (CDR Y))))).
```

This function searches for the first occurrence of X as an element of Y. If found, it returns the CDR of Y following X. Otherwise, evaluation of the function recurses indefinitely.

The theorem prover can prove:

(IMPLIES (MEMBER A B) (LTE (CHOP A B) B)).

If IMPLIES is an EXPR (as all defined functions are under our eval) and has its arguments evaluated before the definition is entered, then some instances of this theorem do not terminate when evaluated. (For example, let A be 1 and B be NIL). All instances which do terminate yield T, which is what the theorem prover establishes.

It is difficult to establish whether an extension is consistent after the introduction of a partial function. The easiest way to decide some cases is to find a total fixed point for the function definition. If one is found, a model exists in which the defining axiom of the function is satisfied, so the extension is consistent.

In practice, the theorem prover proves very few theorems about functions which do not always terminate. This is primarily because there are very few things one can say about all values of an expression which may not always be defined. The situation is more interesting if one can discuss the sequence of computations a function performs when evaluated.

Turning to automatic proof of termination, it is important to point out that the system described here cannot deal with such proofs. If termination is being considered, it is necessary to allow the logic to discuss the notion of being well-defined, or of being undefined. Since our logical language is the programming language itself, we are inherently unable to discuss termination of programs in the language.

From a more mundane point of view, many of the rewrite rules assume termination for certain subexpressions. For example, the rules:

```
(COND x y y) => y,
```

and

```
(EQUAL x x) => T,
```

do not preserve the termination properties of the left-hand side. If x does not terminate, neither of the left-hand sides above terminate when evaluated. But both right-hand sides can terminate. Thus, even if we introduced a new logical language to allow discussion of termination, about half a dozen rewrite rules could be used only after determining whether certain subexpressions terminated.

However, the present approach has an important contribution to make to proof of termination. This is simply the observation that the automatic induction heuristics apply to termination problems as well as to the other properties of programs.

For example, to prove that LTE always terminates for well-defined arguments, we proceed as follows. The definition of LTE is:

```
(LTE (LAMBDA (X Y)
     (COND X
           (COND Y (LTE (CDR X) (CDR Y)) NIL)
           T))).
```

eval determines that induction on both arguments is required. We must then show that if both arguments evaluate, and either is NIL, the result is well-defined. For X = NIL, this is easy. For Y = NIL, we must do a case analysis on X, using the fact that it is well-defined. For the induction step we assume that LTE is well-defined when the values of the arguments are val_1 and val_2. We must show that it is well-defined when the values are well-defined pairs, with val_1 and val_2 as their respective CDRs. Since both X and Y are well-defined and non-NIL, we immediately observe that we must show that:

```
        (LTE (CDR X) (CDR Y))
```

is well-defined. Both arguments evaluate, since X and Y are defined. Furthermore, their values are val_1 and val_2 by construction. But we have assumed that LTE is defined for these arguments, so we are done.

In short, the evaluation mechanism is still very well suited to choosing the terms to induct upon and for determining the form of the induction argument needed to prove termination.

5.2 Iteration

We have not considered iterative functions up to this point. The current program cannot deal with them. However, it is interesting to note that the methods developed in this thesis can be extended to allow proofs about such functions. More precisely, we first observe that iterative functions have natural recursive counterparts -- that is, there are recursive functions which build up or decompose their arguments in the same way that a particular iterative function does. It is possible to mechanically translate a function using PROG, SETQ, GO, and RETURN into its recursive version. This is discussed and illustrated below. We then demonstrate, by example, how it is possible to prove theorems about such "iterative" functions by a slight extension of the induction heuristic and fertilization rule. In particular, it becomes clear that for some theorems at least it is possible to produce an inductive proof by purely mechanical means, even for iterative functions.

Below we give an iterative definition of a function which reverses a list (constructively). The traditional LISP COND is used, along with PROG, SETQ, GO, and RETURN. We assume the reader is familiar with these primitives in LISP:

```
          (PROGREV (LAMBDA (X)
                   (PROG (Y)
                         (SETQ Y NIL)
                    LOOP (COND (X (SETQ Y (CONS (CAR X) Y))
                                  (SETQ X (CDR X))
                                  (GO LOOP))
                               (T (RETURN Y)))))).
```

This function uses Y as a local variable. Y is initialized to NIL and then the program cycles around a loop, CDRing X. The CAR of X is CONSed onto the running answer, Y, until X is empty. Y is returned.

This behaviour is simulated by the following recursive definitions. The three-place COND used elsewhere in this paper is used below.

```
          (PROGREV (LAMBDA (X) (REV1 X NIL)))
```

where the definition of REV1 is:

```
          (REV1 (LAMBDA (X Y)
                (COND X
                      (REV1 (CDR X) (CONS (CAR X) Y))
                      Y))).
```

Here, PROGREV just calls REV1, initializing Y to NIL. REV1 recurses, CDRing X. The CAR of X is CONSed onto the running answer, Y, until X is empty. Y is returned. The relationship with the iterative definition is obvious.

This translation can be done in a straightforward mechanical way. In fact, the optional routine, <u>defineprog</u>, exists which will translate a well-formed definition involving arbitrary PROGs, SETQs, GOs and RETURNs into the appropriate collection of mutually defined recursive functions. <u>defineprog</u> will produce the two definitions above, given the original PROG version. It will also handle more complicated loops and jumps.

The basic idea in <u>defineprog</u> is that one generates functions which have as many variables as there are arguments and locals in the current

PROG. Each label indicates that a new recursive function must be produced. This function will be called whenever a GO statement to the associated label is encountered. SETQs are handled by accumulating the assignments until a GO is encountered. At that point, a call to the relevant function is planted, with the argument positions set to the results of the accumulated assignments to each variable. Thus, in the recursive call, the variables are bound as desired. RETURNs indicate that the process is done. The reader should now be able to construct REV1 from the PROG version of PROGREV.

We now wish to illustrate how we can prove theorems about these recursive translations of iterative functions, without requiring user supplied inductive assertions. We will prove the equivalence of REVERSE (the traditional definition) and PROGREV (the recursive translation of the iterative version). The theorem to be proved is:

```
          (EQUAL (REVERSE A) (PROGREV A)).
```

This just becomes:

```
(1)       (EQUAL (REVERSE A) (REV1 A NIL)),
```

by normalation, since PROGREV is not recursive.

We shall approach the proof just as the current program does, and point out the necessary extensions when they are encountered.

The evaluation routine indicates that induction on A is required. The NIL-case evaluates to T. We assume (1) as our induction hypothesis, and set out to prove:

```
          (EQUAL (REVERSE (CONS A1 A)) (REV1 (CONS A1 A) NIL)).
```

This just evaluates to:

```
(2)       (EQUAL (APPEND (REVERSE A) (CONS A1 NIL))
                 (REV1 A (CONS A1 NIL))).
```

Cross-fertilizing with the induction hypothesis, (1), by replacing (REVERSE A) by (REV1 A NIL), we obtain:

```
(EQUAL (APPEND (REV1 A NIL) (CONS A1 NIL))
       (REV1 A (CONS A1 NIL))).
```

Since (CONS A1 NIL) is common to both sides, we generalize it:

```
(EQUAL (APPEND (REV1 A NIL) B) (REV1 A B)).
```

This is an interesting lemma about APPEND and REV1.

Up to this point we have used only techniques available in the current program. However, in setting up the induction hypothesis to prove the lemma above, we diverge slightly, but in a very mechanical way.

Let us consider the induction principle. If we are trying to prove p(X,Y) where X and Y are universally quantified, and wish to use induction on X, a valid induction step is:

$$\forall X(\forall Y(p(X,Y)) \rightarrow \forall Y,X1(p((\text{CONS } X1\ X),Y))).$$

This corresponds to taking as our induction hypothesis, the expression p(A,Y), where Y is a free variable, and trying to prove p((CONS A2 A),B). That is, we get to assume that p is true for A and all Y, and we must prove that it is true for (CONS A2 A) and B. Intuitively, the "opponent" gets to choose A, A2, and B, and we know nothing about them; but we can choose any Y we like. Indeed, we can use as many instances of p(A,Y) as we wish.

In the past we have predicted that the only instance we will want to use is p(A,B) and assumed it explicitly. This could be predicted because none of the functions CONSed things on to their arguments during recursion, so that in the conclusion, B would remain unchanged. This is not the case with REV1 and the extension desired is clear: If when evaluation is determining what to induct upon, it is noted that skolem constants are being built up in the recursion, then let those skolem constants be free in the hypothesis (provided they are not being inducted upon).

Returning to the REV1 theorem above, we therefore choose as our induction hypothesis:

```
(3)       (EQUAL (APPEND (REV1 A NIL) Y) (REV1 A Y)),
```

where Y is a free variable. The conclusion to be established is:

```
          (EQUAL (APPEND (REV1 (CONS A2 A) NIL) B) (REV1 (CONS A2 A) B)).
```

Using the program's approach, we evaluate this and get:

```
(4)       (EQUAL (APPEND (REV1 A (CONS A2 NIL)) B) (REV1 A (CONS A2 B))).
```

As usual, we must now use our hypothesis, (3). Since it has Y free in it, we can use it in two places and choose to do both. The first is done by letting Y be (CONS A2 B) and fertilizing the right-hand side of (4) with (3). This produces:

```
(5)       (EQUAL (APPEND (REV1 A (CONS A2 NIL)) B)
                 (APPEND (REV1 A NIL) (CONS A2 B))).
```

The second use of (3) is by letting Y be (CONS A2 NIL), and fertilizing the left-hand side of (5) with (3):

```
(6)       (EQUAL (APPEND (APPEND (REV1 A NIL) (CONS A2 NIL)) B)
                 (APPEND (REV1 A NIL) (CONS A2 B))).
```

Having made these two non-standard fertilizations, we now proceed as the program would.

Noting that (REV1 A NIL) is common to both sides, we generalize (6) to get:

```
          (EQUAL (APPEND (APPEND C (CONS A2 NIL)) B)
                 (APPEND C (CONS A2 B))).
```

This is just a trivial lemma about APPEND which the current theorem prover can establish immediately, by induction on C. We are therefore finished, and have shown that PROGREV is equivalent to REV.

The extensions necessary to produce this proof were: (1) Induction had to notice that the skolem constant B was being built up and leave it free in the hypothesis. (2) Fertilization had to use two different instances of the induction hypothesis.

The proof had exactly the same structure as those produced by the program. In particular, fertilization and generalization produced the necessary lemmas. This kind of extension is sufficient to allow proofs of several other theorems involving iterative functions.

As a topic of further research, the theorem prover is being modified to handle such functions. However, allowing free variables in the hypothesis introduces many problems. In particular, it allows the hypothesis to be used in many ways, not all of which are good. Note that in the proof just described, the result of the second fertilization, (6), can again be fertilized. In fact, because an instance of the right-hand side of (3) occurs in the left-hand side of (3), fertilization with this equality can continue indefinitely. Furthermore, if free variables are introduced into induction hypotheses, then quantification gets messy if induction must be resorted to a second time. Both problems can be avoided if <u>eval</u> is used to predict exactly which instances will be needed, and then have <u>induct</u> supply just those instances. This is the direction of current research.

CHAPTER 6 CONCLUSIONS

6.1 Built-in Information

The program "knows" a lot about the primitives of LISP, about how to determine whether certain expressions have certain properties, about how to use an inductive argument to write recursive functions, and about how to use its knowledge of recursive functions to produce inductive arguments. However, no information is built-in about non-primitive functions. This statement is discussed and amplified below.

Although EQUAL is a primitive function, it can be written in terms of the other primitives:

```
(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)))).
```

In addition to being able to prove that EQUALP is symmetric, reflexive, and transitive, the program can prove that it is equivalent to EQUAL:

```
(EQUAL (EQUALP A B) (EQUAL A B)).
```

This is done by breaking the theorem into two conjuncts:

```
(COND (COND (EQUALP A A) T (*1))
      (COND (EQUALP A B) (EQUAL A B) T)
      NIL),
```

with normalation and fertilization, and then normalation again. The first conjunct requires EQUALP to be reflexive, and the second requires that it implies equality. Thereafter, fertilization is not used in this theorem.

The first conjunct is proved by CAR and CDR induction on A, and the second by CAR and CDR induction on A and B simultaneously. The theorem requires 9 seconds.

It should be noted that many of the rewrite rules applied to EQUAL expressions behave as though the expressions were really EQUALP expressions. In particular, _eval_ expands

    (EQUAL (CONS A B) (CONS C D))

in exactly the way it recursively expands

    (EQUALP (CONS A B) (CONS C D)).

Also, _normalize_ rewrites (EQUAL x NIL) to (COND x NIL T), in the same way that normalation would rewrite (EQUALP x NIL). So in many senses, EQUAL is not as built-in as it might appear.

However, certain critical information is available about EQUAL which is not available about EQUALP. This of course is found in fertilization and equality substitution. The theorem prover "knows" that if (EQUAL A B) is T, then A = B. The only sense in which it "knows" this fact for (EQUALP A B) is in being able to prove the theorem stating the equivalence of EQUALP and EQUAL.

Since EQUAL is the characteristic function of the only predicate in the theory, namely, equality, it was felt that it should be primitive.

The only knowledge the program has of non-primitives is their definitions. In fact, only _eval_ and the typing functions even know of the existence of definitions for non-primitives.

A review of the description of the program will reveal only two places in which non-primitives are mentioned outside of function entry in _eval_. These two places are generalization and induction. Both may introduce the non-primitive IMPLIES, and _induct_ may introduce AND. Of course, both of these routines could use the equivalent compound COND statements into which these functions expand. Since normalation immediately removes the IMPLIES and AND, their use by these two routines is actually an inefficiency. It is tolerated because it makes the output of these routines more readable to the user.

A more interesting use of non-primitives occurs elsewhere in generalization, specifically, in _typeexpr_. As noted, this function makes explicit, programmed checks for types BOOLEAN and NUMBERP, before bothering to generate new type functions. It could be argued that information about these two non-primitives is therefore critically built-in.

But as pointed out, without these explicit checks, _typeexpr_ will write its own (identical) definitions of these functions. (If NUMBERP did not exist, _typeexpr_ would invent it.) Since no non-definitional properties of these two functions are known, it hardly matters that the type functions introduced have the names BOOLEAN and NUMBERP, or FOOTYPE and BARTYPE.

Of course, the most interesting use of non-primitives by the program is when it automatically writes a new function to help a generalization. This involves no built-in information about any specific functions, but rather general knowledge of the types of the primitives and the relationship between recursion and induction.

Introducing the logical connectives, AND, OR, NOT, and IMPLIES, as non-primitives to a theorem prover may appear to carry the ban on programmed non-primitive information a little too far. However, since all of these are naturally defined with COND, and since the relevant properties of COND must be known to the theorem prover anyway, introducing special logical facilities would only add more code and produce an interface problem between the logical rules and the knowledge of LISP.

The program uses no lemmas whatsoever concerning user defined functions. (This holds even for AND, OR, NOT, and IMPLIES.) Because no such lemmas are used, the system frequently reproves facts. A very common such fact is that APPEND is associative. This is proved as a lemma for several theorems (a process requiring 3 seconds).

Lemmas are avoided for three reasons. The first is simply that a (malicious) user could introduce FOOAPPEND in a theorem and any time spent by the program looking for applicable lemmas would be wasted, even though the proof of the theorem is just as obvious as before. Secondly, the ability of the program to automatically generate nice lemmas is accentuated by the fact that none of these lemmas are built-in. Finally, one of the primary aims of this project has been to demonstrate clearly that it is possible to prove program properties entirely automatically. A total ban on all built-in information about user defined functions thus removes any taint of user supplied information.

6.2 Automatic Generation of Natural Lemmas

One of the most striking features of proofs produced by the program is the frequency with which natural lemmas are automatically generated. The earliest example of this was in Section 2.3 in which the theorem:

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

was proved. During the proof the program generates the two lemmas:

```
(EQUAL (APPEND C NIL) C),
```

and

```
(EQUAL (APPEND C (APPEND D E)) (APPEND (APPEND C D) E)).
```

The first states that NIL is a right identity for APPEND, and the second states that APPEND is associative.

Further examples are provided in Chapter 4. The proof of

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

produces the two lemmas:

```
(IMPLIES (ORDERED C) (ORDERED (ADDTOLIST B C)))
```

and

```
(OR (LTE B D) (LTE D B)).
```

Both of these are very basic properties of the functions concerned, although that fact is certainly not known to the system. The lemmas are generated entirely automatically by the combination of induction and the generalization and normalation routines.

Proof of the theorem in Section 4.5:

```
(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A))),
```

generates:

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

as a lemma, which was the first theorem discussed in this section.

As a final example of how natural lemmas are produced, consider the program's proof that TIMES (multiplication defined in terms of addition) is associative:

```
(EQUAL (TIMES A (TIMES B C)) (TIMES (TIMES A B) C)).
```

An induction on A produces the induction step:

```
(COND (EQUAL (TIMES A (TIMES B C))
             (TIMES (TIMES A B) C))
      (EQUAL (PLUS (TIMES B C) (TIMES A (TIMES B C)))
             (TIMES (PLUS B (TIMES A B)) C))
      T),
```

after normalation. Cross-fertilization produces:

```
(COND (EQUAL (PLUS (TIMES B C) (TIMES (TIMES A B) C))
             (TIMES (PLUS B (TIMES A B)) C))
      T
      (*1)).
```

Ignoring the (*1) term and generalizing produces:

```
(EQUAL (PLUS (TIMES B C) (TIMES D C))
       (TIMES (PLUS B D) C)),
```

which is just the theorem that TIMES distributes over PLUS.

This is proved by induction on B. The normalated induction step is:

```
(COND (EQUAL (PLUS (TIMES B C) (TIMES D C))
             (TIMES (PLUS B D) C))
      (EQUAL (PLUS (PLUS C (TIMES B C)) (TIMES D C))
             (PLUS C (TIMES (PLUS B D) C)))
      T).
```

Cross-fertilization and generalization produce:

```
(EQUAL (PLUS (PLUS C E) F) (PLUS C (PLUS E F))),
```

(ignoring the (*2) term). This is just the associativity of PLUS.

The ability of the system to automatically generate critical lemmas accounts for a large degree of its success. This ability is largely a product of the design philosophy discussed in the next section.

6.3 Design Philosophy of the Program

The program was designed to behave properly on simple functions. The overriding consideration was that it should be automatically able to prove theorems about simple LISP functions in the straightforward way we prove them. This, I believe, has been achieved. The principle is clearly reflected in the following areas:

Evaluation is of monumental importance to the system. This routine was designed to perform the task of "stepping through" the evaluation of a function. It cautiously goes into recursion, and notices what structures are being decomposed, what is necessary for the function to run properly, and how the arguments are used. It returns a symbolic expression describing the result of the evaluation and notes where recursive functions halted and why. It is very natural that the backbone of a theorem prover for LISP should be an interpreter for LISP. Furthermore, most of the intuitions that a human programmer/theorem prover possesses about LISP comes directly from the behaviour of the interpreter upon given expressions.

The second area in which the desire for autonomy and a sound, simple approach is evident is the typing function, <u>typeexpr</u>. The very existence of this function (as opposed to, say, user interaction) strongly shows the influence of the above design principle. The program has the means to discover for itself the kind of output an expression may yield. It is flexible enough to express this as an automatically produced description of the output, rather than merely naming one of a predetermined number of type classes. Finally, its orientation is such that this description is in the form of a LISP program which returns T or NIL according to membership in the defined class, rather than in some other language.

In addition to the existence of <u>typeexpr</u>, its methods are in accordance with the principle expressed above. It generates its description by recursively exploring the expression, and it recognizes when a recursive description is called for by using an inductive approach. Furthermore, once its first pass has produced a rather sloppy piece of LISP code which will do the job, it uses its extensive knowledge of the language to optimize it. This is much like a programmer. It allows the production of efficient code, known to work, without the aid of outside help or search.

A third area reflecting the influence of the desire to approach theorems as a human might is induct. Again, its existence alone is indicative of the spirit of the program. Most programmers merely "observe" that APPEND is associative, or that some fact about SORT holds. They are not given hints regarding how to understand that a particular piece of recursion has a certain property. Neither do they require each statement in a program to be accompanied by a description of what it does; presumably that is unambiguously clear in a programming language. Their "observation" of a property is apparently the result of running the function symbolically, with enough control to know when the recursive calls are satisfied by an induction hypothesis. This is precisely how the program works, using eval to inform induct how the functions are running and what inductive "patches" are needed to handle the recursion.

The structure of the program is remarkably simple by artificial intelligence standards. This is primarily because the control structure is embedded in the syntax of the theorem. This means that the system does not contain two languages, the "object language", LISP, and the "meta-language", predicate calculus. They are identified. This mix of computation and deduction was largely inspired by the view that the two processes are actually identical. Bob Kowalski, Pat Hayes, and the nature of LISP deserve the credit for this unified view. One implication of its use here is that there are no communication or interface problems between a LISP knowledge system and a logic knowledge system. This not only helps reduce the program's size and complexity, but increases its power. Although no analogy of method is intended, such a complete integration of programming skills and logical reasoning is certainly present in a good programmer.

It was noted that several of the routines used the same rewrite rules. This is a design feature, and partially accounts for the speed of the program. It is far more efficient to check frequently whether the first argument of a COND is a CONS, than it is to wait and let *eval* do it. This is a product of the view that the program's knowledge should be as integrated as possible, and that fast powerful simplification routines should be used before any more sophisticated techniques are used.

Finally, it should be pointed out that the program uses no search. At no time does it "undo" a decision or back up. This is both the primary reason it is a fast theorem prover, and strong evidence that its methods allow the theorem to be proved in the way a programmer might "observe" it. The program is designed to make the right guess the first time, and then pursue one goal with power and perserverance.

APPENDIX A FUNCTION DEFINITIONS

```
(ADD1 (LAMBDA (X) (CONS NIL X)))

(ADDTOLIS (LAMBDA (X Y)
          (COND Y
                (COND (LTE X (CAR Y))
                      (CONS X Y)
                      (CONS (CAR Y) (ADDTOLIS 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 X (CDRN (SUB1 X) (CDR Y)) Y) NIL)))

(CONSNODE (LAMBDA (X Y) (CONS NIL (CONS X Y))))

(CONSTTRU (LAMBDA (X) T))

(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 X (ADD1 (ADD1 (DOUBLE (SUB1 X)))) 0)))

(ELEMENT (LAMBDA (X Y)
         (COND Y (COND X (ELEMENT (CDR X) (CDR Y)) (CAR 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 X (NOT (EVEN1 (SUB1 X))) T)))

(EVEN2 (LAMBDA (X)
       (COND X (COND (SUB1 X) (EVEN2 (SUB1 (SUB1 X))) NIL) T)))

(FLATTEN (LAMBDA (X)
         (COND
            (NODE X)
            (APPEND (FLATTEN (CAR (CDR X))) (FLATTEN (CDR (CDR X))))
            (CONS X NIL))))

(GT (LAMBDA (X Y)
    (COND X (COND Y (GT (SUB1 X) (SUB1 Y)) T) NIL)))

(HALF (LAMBDA (X)
      (COND X (COND (SUB1 X) (ADD1 (HALF (SUB1 (SUB1 X)))) 0) 0)))

(IMPLIES (LAMBDA (X Y) (COND X (COND Y T NIL) T)))

(INTERSEC (LAMBDA (X Y)
          (COND X
                (COND (MEMBER (CAR X) Y)
                      (CONS (CAR X) (INTERSEC (CDR X) Y))
                      (INTERSEC (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 X (COND Y (LTE (SUB1 X) (SUB1 Y)) NIL) T)))

(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)))

(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))))

(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)))

(PLUS (LAMBDA (X Y)
      (COND X (ADD1 (PLUS (SUB1 X) Y)) (LENGTH Y))))

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

(SORT (LAMBDA (X)
      (COND X (ADDTOLIS (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)))

(TIMES (LAMBDA (X Y)
       (COND X (PLUS Y (TIMES (SUB1 X) Y)) 0)))

(TIPCOUNT (LAMBDA (X)
          (COND
            (NODE X)
            (PLUS (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)))
```

APPENDIX B SOME THEOREMS PROVED AUTOMATICALLY

Theorems about APPEND, REVERSE, and LENGTH

```
(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)))
```

Theorems about membership

```
(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 (INTERSEC 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 (INTERSEC A B) A))

(EQUAL (MEMBER A B) (NOT (EQUAL (ASSOC A (PAIRLIST B C)) NIL)))
```

Theorems about MAPLIST

```
(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))
```

Theorems about miscellaneous functions

```
(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 A (EQUAL B C)) (EQUAL (EQUAL A B) C)))
```

Theorems about arithmetic functions

```
(EQUAL (PLUS A B) (PLUS B A))

(EQUAL (PLUS A (PLUS B C)) (PLUS (PLUS A B) C))

(EQUAL (TIMES A B) (TIMES B A))

(EQUAL (TIMES A (PLUS B C)) (PLUS (TIMES A B) (TIMES A C)))

(EQUAL (TIMES A (TIMES B C)) (TIMES (TIMES 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) (TIMES 2 A))

(EQUAL (DOUBLE A) (TIMES A 2))

(EQUAL (EVEN1 A) (EVEN2 A))
```

Theorems about ordering relations

```
(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)
```

Theorems about functions which inspect tree-structured lists

```
(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 FOUNDATIONS II

We wish to demonstrate the consistency of list theory by exhibiting it as a conservative extension of number theory. Results derived in Shoenfield 1967 establish that any extension of number theory by total,recursive functions is a conservative extension. Thus, by exhibiting total, recursive definitions of the primitives in list theory, we will have established that list theory is a conservative extension of number theory. Furthermore, this automatically means that any extension of the theory of lists by total, recursive functions is a conservative extension.

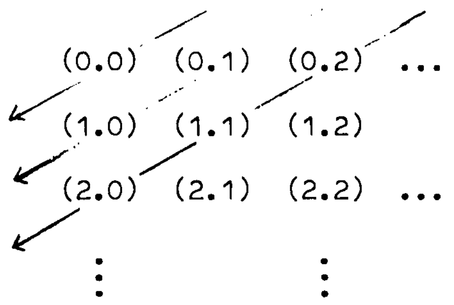
In the following, LISP notation will be used to denote function application. However, the usual number theoretic operations, such as + and $<$, will be denoted by the traditional infix notation.

All abbreviations introduced into list theory will be inoperative below to prevent confusion. In particular, we will be discussing the natural numbers, N; any occurrence of an integer will denote an element of N, not a list of NILs.

We will assign an element of N to every distinct specific list. This is done by letting NIL be (an abbreviation for) 0. CONS will be a "pairing function", which maps elements of N x N into the non-zero elements of N. The definition is:

$$(\mathrm{CONS}\ X\ Y) = \frac{(X + Y)\ (X + Y + 1)}{2} + X + 1.$$

This is just the traditional diagonal enumeration of pairs:



That is,

(CONS NIL NIL) = (CONS 0 0) = 1,

(CONS NIL (CONS NIL NIL)) = (CONS 0 1) = 2,

(CONS (CONS NIL NIL) NIL) = (CONS 1 0) = 3,

etc.

It should be noted that:

X < (CONS X Y) & Y < (CONS X Y).

CAR and CDR are just the functions which map from N to N such that:

$X \neq 0$ -> (CONS (CAR X) (CDR X)) = X.

The recursive definitions of CAR and CDR are derived in terms of the diagram above. Call the point (x.y) the kth point if k = (CONS x y).

Let (DIAGCNT K) be the number of complete diagonals up to (and possibly including) the one passing through the Kth point. The definition is:

$$\text{(DIAGCNT K)} = \mu \text{N}((\sum_{\text{I}=0}^{\text{N}+1} \text{I}) > \text{K}).$$

For notational simplicity, let (POINTCNT K) be the total number of points on complete diagonals up to (and possibly including) the one passing through point K:

$$\text{(POINTCNT K)} = \sum_{\text{I}=0}^{\text{(DIAGCNT K)}} \text{I}.$$

Then consideration of the diagram above shows that CAR and CDR depend upon whether the point in question is at the end of its diagonal, upon the number of complete diagonals up to (and possibly including) the one through the point, and upon the number of points on the given point's diagonal before the given point. The definitions of CAR and CDR are:

$$(\text{CAR K}) = \begin{cases} (\text{DIAGCNT K}) \dot{-} 1, \text{ if } K = (\text{POINTCNT K}), \\ (K \dot{-} (\text{POINTCNT K})) \dot{-} 1, \text{ otherwise.} \end{cases}$$

$$(\text{CDR K}) = \begin{cases} 0, \text{ if } K = (\text{POINTCNT K}), \\ (\text{DIAGCNT K}) \dot{-} (K \dot{-} (\text{POINTCNT K})) + 1, \text{ otherwise.} \end{cases}$$

The definitions of EQUAL and COND are:

$$(\text{EQUAL X Y}) = \begin{cases} 1, \text{ if } X = Y, \\ 0, \text{ otherwise.} \end{cases}$$

$$(\text{COND X Y Z}) = \begin{cases} Z, \text{ if } X = 0, \\ Y, \text{ otherwise.} \end{cases}$$

In view of the total, recursive definitions of all of the functions introduced, adding them to number theory produces a conservative extension. However, in this extension, it is possible to derive, as theorems, the non-logical axioms of the theory of lists. This is straightforward. As shown below, the induction schema is also a theorem in this extension. Thus, this extension represents the theory of lists as a conservative extension of number theory. Therefore, not only is it consistent, but any extension of it by a total, recursive function is a conservative extension.

The induction schema (of list theory) is now:

$$p(0) \;\&\; \forall X,Y(p(X) \;\&\; p(Y) \;\rightarrow p((\text{CONS X Y}))) \;\rightarrow \forall X(p(X)),$$

where X and Y are quantified over the elements of N, and CONS is the number theoretic function defined above.

We wish to show that this is a theorem in the extension. Therefore, assume that the hypothesis is true, but the conclusion (that p(X) is true for all X) is false. Then by the least number principle of number theory, there is a smallest K such that p(K) is false. Let this K be K_0.

If K_0 is 0 the hypothesis is violated, since p(0) is true. Therefore, $K_0 \neq 0$.

In this case, we know that:

$$K_0 = \text{(CONS (CAR } K_0\text{) (CDR } K_0\text{))},$$

and that:

$$\text{(CAR } K_0\text{)} < K_0 \quad \& \quad \text{(CDR } K_0\text{)} < K_0.$$

Since (CAR K_0) and (CDR K_0) are both strictly less than K_0, and since K_0 is the smallest K such that p(K) is false, we know that:

$$p(\text{(CAR } K_0\text{)}) \quad \& \quad p(\text{(CDR } K_0\text{)}).$$

But since the hypothesis of (1) is true, we get:

$$p(\text{(CONS (CAR } K_0\text{) (CDR } K_0\text{))}),$$

or $p(K_0)$. But this is a contradition. Therefore, the right-hand side of (1) must be true, and the induction schema holds in the extension.

Since the theory of lists can be represented as a conservative extension of number theory, we know any total, recursive function can be added to produce a conservative extension. At this point we can rely upon any of the well-known recursive schemas. One such schema is:

```
(f X Y) = (COND X
                (h (f (CAR X) Y) (f (CDR X) Y) X Y)
                (g Y)),
```

where h and g are total, recursive functions. This is just a variant of course-of-values recursion.

APPENDIX D COMPARATIVE SURVEY OF OTHER WORK

There are four commonly used inductive methods for proving properties of programs. These are: computational induction (Park 1969, and deBakker and Scott 1969), structural induction (Burstall 1969), recursion induction (McCarthy 1963), and inductive assertions (Naur 1966, Floyd 1967, and Manna 1969). The first three of these are primarily concerned with recursive functions, while the fourth deals with iterative processes and assignment. Floyd's method has been generalized to handle recursive functions by Manna and Pnueli (1970). The first and third methods are essentially induction on the depth of function calls. The second is induction on the data structures being altered, and the fourth is induction on the length of the computation path. As shown in Manna, Ness, and Vuillemin 1972, these methods are all essentially equivalent.

The paper by Manna, et al. 1972 is a very readable introduction to the various inductive methods. The volume containing that paper presents a good cross-section of the current work. It should be pointed out that many of the theorems cited in the literature as illustrative examples of a particular proof technique, have been proved automatically by the program described here. In particular, proofs of properties of simple functions such as APPEND and REVERSE are generated by this program. However, it should also be said that many other theorems cited in the literature are beyond the capabilities of the current program -- particularly those dealing with termination arguments or functions which recurse "up" rather than "down".

The most commonly used inductive method is that of Floyd. In this approach, assertions are attached to key points (such as loops) in the flow-diagram, and an assertion must be true each time control passes through the relevant point. Verifying the correctness of the program consists of proving that, for each path through the diagram, each assertion implies the next one in the path provided the effects of the intervening program statements are taken into account. Manna 1969 describes a similar method used to prove termination. This method may be thought of as attaching assertions to points such that the assertion is true during some pass through the point, rather than all passes as in Floyd's method.

All of the implemented systems which use Floyd's methods assume the assertions are supplied by the user. Recent work by Wegbreit (1973) and Katz and Manna (1973) present some heuristics for automatically generating inductive assertions using the entry/exit conditions required and the program text. The automatic generation of induction formulas and lemmas by our program is equivalent to the automatic generation of assertions for Floyd's method.

The inductive approach used by the program described here is structural induction. Burstall's 1969 paper gives an excellent account of this technique and presents several proofs similar to those automatically generated by this program. Logicians frequently use structural induction to establish meta-theorems, by inducting upon the structure of formulas in the theory. This was used several times in this paper. Curry and Feys (1958) named the method, and McCarthy and Painter (1967) used it to establish the correctness of a compiler. Burstall points out the similarity between the form of the proofs and the form of the recursive functions involved. However he does not explicitly describe heuristics for choosing what to induct upon, or how to generate the induction formula.

There is, of course, no reason why one should feel restricted to one inductive method over another. Each has its own merits and disadvantages. For example, in the presence of assignments to data structures, Floyd's methods have been shown to be applicable. But since circular structures exist in such systems, structural induction does not readily apply. (There are objects which cannot be constructed by CONSing atoms together. In fact, there are structures which contain no atoms at all. Burstall (1972) discusses the problems of destructive assignment and the unique problems it presents.) Thus, one's approach to the problem of proof of program properties largely depends upon the features of the particular language discussed and the theorems at hand.

We will now review other implemented systems and make some comparisons with the program described here. The primary points of interest are the aims and methods of the system, and the features of the languages used to present programs and assertions (or theorems). Several general remarks are in order.

The only published references to an implementation of computational induction and recursive functions are those of Milner (see references below). The remaining systems are concerned with Floyd-like methods for flow-diagram languages.

It should be noted that except for King and Cooper, all of the systems were designed to provide mechanical aid to a human verifier. They concentrate on the generation of the verification conditions given the program and the attached assertions, algebraic and logical simplication, and bookkeeping services during the human's proof. Their automatic theorem proving capabilities are purposely limited or nonexistent.

No previous system (known to this author) has automated the "creative" parts of the proof process, namely: generalization of the theorem to be proved, application of automatic program writing to simplify the problem, and the automatic generation of induction formulas. Automation of these parts of the process has been the primary goal of the project reported here.

Milner's 1972 papers describe his implementation of Scott's Logic for Computable Functions (LCF) (Scott 1970). He shows how the syntax and semantics of a programming language allowing assignment, conditionals, while statements, and compound statements may be expressed in LCF. His program is an LCF proof-checker. The basic induction rule is computational induction, although a proof of the recursion induction rule is presented as an example of a theorem proof-checked by the program. The program accepts expressions in LCF as theorems to be proved, and then obeys commands from the user directing the application of the rules of inference. The program keeps track of the goals to be established and the steps carried out in each proof.

LCF is designed for handling functions of higher type computed by arbitrary recursive programs. The system is therefore quite capable of handling programs which may not terminate, and, unlike any other system currently available, programs which deal with other programs as arguments and results. This is probably the most powerful and flexible verification system implemented. As noted above, this power is completely controlled and directed by the user.

In Milner and Weyhrauch 1972, the authors describe the use of the LCF proof-checker to verify the correctness of a compiler. The authors state that parts of many proofs followed patterns that appeared to be amenable to complete automation. Those parts not so amenable were, predictably, the selection of the induction formulas.

King 1969 and Good 1970 deal with ALGOL-like flow-diagram languages with Floyd-like methods. They allow integer valued variables and one dimensional arrays with integer elements. Statements include assignments to variables and array elements, conditional statements, and jump statements. The usual arithmetic functions may be used in assignment statements, and the usual boolean arithmetic relations are allowed in the tests in conditionals. Procedure calls are not allowed.

King's program was designed as a fully automatic system for the Floyd method. Once the user has submitted his program text with assertions, King's system constructs the verification conditions and then tries to prove them. Proof is carried out by an arithmetic theorem prover designed specifically for this task. Knowledge of the arithmetic functions and predicates is built-in.

An assertion is just a boolean ALGOL statement, with universal and/or existential quantification. It is not possible to introduce a procedure to express an assertion. Thus, assertions are restricted to expressions compounded from the primitive boolean relations and arithmetic functions. This severely limits the expressive power of the assertion language. For example, functions such as summation and greatest common divisor are not built-in, and thus, not available.

In order to make them available, new routines (in assembly code) must be written. This means that the system will contain two very large and completely independent knowledge bases, one about ALGOL procedures, and one about mathematical functions and relations. In particular, the extensive knowledge of ALGOL cannot be used to "understand" a mathematical function, given an ALGOL procedure for computing it. Thus, in building-in a new function, all of the facts which may ever be needed about the properties of the new function and its relationships with the existing primitives must also be built-in.

King's system has automatically verified several interesting programs, including an array sorting program and a program which raises an integer to a power using the binary representation of the power.

Good's program generates verification conditions from the user supplied text and assertions. However, it makes no attempt to automatically prove them. Because the machine was not designed to "understand" the assertions, the assertion language is maximally flexible: it is simply arbitrary text strings, usually expressing relations in English. The program only recognizes occurrences of program variables in these strings, and forms verification conditions by substituting for them, according to the semantics of the intervening statements. It informs the user of the conditions to be verified and will perform a bookkeeping service as the user convinces himself of the validity of the conditions.

Cooper 1971 presents a theorem prover that deals with flow-diagram languages like those above, without provision for arrays. Programs are based on block structures, and blocks are allowed to contain sub-blocks. The usual integer relations and functions are again available. The program is designed to automatically generate and prove the termination conditions for flow-diagrams. It recognizes simple counting loops, and under certain conditions of linearity, will automatically generate a termination condition for such a loop. If the condition cannot be generated, the user must supply one. These conditions are always logical formulas compounded from the primitives. New functions cannot be expressed. Once termination conditions have been produced, either automatically or manually, they are passed to an automatic arithmetic theorem prover based on the Presburger algorithm.

Gerhart 1972 describes the use of Floyd's methods on an APL subset. Her assertion language is APL, but again, she allows no defined procedures, either in the program or in assertions. The only process automated is the verification of the compatibility of argument types and APL operators.

Topor and Burstall (1973) have implemented a Floyd-like system using symbolic evaluation. They deal with an ALGOL-like language allowing integer arithmetic. Recursive procedures are allowed. They attach inductive assertions to points in the program; these assertions are expressed as alternative programs in the language, usually involving higher-level primitives. For example, a program for raising an integer to an integer power using repeated multiplication has inductive assertions using the exponentiation operator of the language. They observe that by symbolically evaluating a program around a given path to a point, the description of the program state upon reaching that point is generated. As in our system, they use symbolic evaluation to produce a case analysis of conditions to be proved. This case analysis is expressed in a separate language. Although some automatic theorem proving is currently done, they intend to automate as much of it as possible in a separate theorem proving stage.

The field is very active and there are many systems currently being implemented about which very little has been published. Among these is a Floyd-like system by Peter Deutsch.

From a programming language point of view, the features of the LISP subset in the system described here are as follows: Arbitrary (non-circular) data structures constructed from list cells are allowed. Besides the four primitives for constructing and accessing list cells, a conditional statement and equality function are provided. The user may introduce any number of recursive functions defined in terms of these primitives or other recursive functions. Theorems are boolean LISP statements, possibly involving non-primitive functions, but restricted to universal quantification. The program is designed to be a fully automatic theorem prover, but does not deal with termination problems. Any instance of a theorem proved is true, provided the functions involved terminate for that instance. The program requires no user supplied assertions other than the statement of the theorem to be proved. Structural induction is the inductive method used.

The flexibility on data structures means that integers, linear lists, and tree structures are trivially available. Functions for accessing the n^{th} element of a list, and for changing the n^{th} element (by copying the initial segment) can be added by the user; this simulates arrays. The flexibility on the introduction of non-primitive functions means that the traditional arithmetic functions and relations are available (since they are recursively computable), as well as many other functions. Since new relations are added as LISP functions, their properties are derived by the system as needed rather than being built-in.

Although the language prohibits PROG, SETQ, GO, and RETURN, an optional routine is available which converts function definitions using these primitives into a set of mutually defined recursive functions.

However, the language also prohibits the concept of an address and assignment to it. This is basic to King and Good, and accounts for a good deal of the complexity of their systems. In the LISP subset allowed here, there is no way to detect the difference between two EQUAL lists, regardless of how they were constructed. (The LISP primitive EQ, which detects equivalence of machine addresses, is not available.) Destructive assignments, RPLACA and RPLACD, are not available either. As a particularly horrid example of the problems involved, let REV be a LISP function which reverses a list by rearranging the pointers in the list cells. Then

```
(EQUAL (REV X) (REV X))
```

does not, in general, evaluate to T. In evaluating the first (REV X), the contents of X are destructively altered, so that when the second (REV X) is evaluated, X represents a different list.

In summary, our system and Milner's are the only ones which allow tree structured data. (Milner even allows functions as data objects.) The arrays allowed by King, Good, and Gerhart are restricted to integer elements. They correspond to linear lists. Of course, the integers are rich enough to simulate trees, but without the necessary primitives the enumeration schemes are prohibitively complicated.

Iteration and assignment to variables can be adequately simulated by recursive function calls. Therefore, the distinquishing features of the various programming languages are whether they allow subroutine calls and/or destructive assignment.

Our system, Milner's, and Topor and Burstall's are the only ones allowing an easily extended assertion or logical language. This just reflects the degree to which this language is restricted to the built-in primitives.

Our system and those of King and Cooper are the only ones encorporating automatic theorem provers. However, ours is the only one which attempts to solve the induction problem directly, by including induction as an automatic rule of inference. The other two theorem provers are restricted to arithmetic, and use algorithms which are capable of deciding many problems in this domain. Without such decision procedures, these problems would require induction to solve.

Bledsoe (1971) encorporated an induction rule in a set theory theorem prover. However, its use was triggered by the occurrence in the theorem of a statement of the form "for all natural numbers, n", and caused induction on n. Such a trigger in our system could cause induction on C in:

(TIMES A (TIMES B C)) = (TIMES (TIMES A B) C),

even though C is not being recursed upon.

Brotz and Floyd (1973) have implemented an automatic theorem prover for arithmetic which is remarkably similar to ours. The user may introduce non-primitive functions by adding the defining equations. These equations may use successor, predecessor, and other defined functions, and recursion. The system uses induction as a rule of inference and generalizes common subterms as described here (although type functions do not exist). However, no equivalent of the bomb list exists. The term chosen for induction is always the right-most argument appearing in the theorem (this puts a trivial syntactic constraint on the order of the arguments to recursive functions). No attempt is made to analyze the recursive structure of the functions involved. This heuristic will not always choose the "right" term to induct upon. For example, if we alter the rule to choose the left-most so that the conventions used in the functions introduced here are respected, the system would attempt induction on A in the theorem:

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

Although A is recursed upon, it is not at all clear how a proof by induction on A would proceed. The program described here would choose B, on the grounds that it is recursed upon more often; as we have seen, this allows a straightforward proof.

The Brotz and Floyd program does not currently handle induction on multiple terms simultaneously, or induction requiring two or more bases. These would be trivial additions (in the sense that they are completely compatible with the existing heuristics). Many of the proofs produced by their system are identical to the proofs of the same theorems by our system.

Returning to the program verification systems discussed, I feel they are noticeably less ambitious than we have been in their desire to automate the processes involved. This thesis has demonstrated that for a large class of functions it is possible to fully automate the inductive proofs of non-trivial properties, without the aid of user supplied assertions or information.

Our work is nicely complemented by (and complements) the work of Darlington 1972. Darlington discusses a system which accepts a recursive function definition and translates it into an equivalent flow-diagram. The input expressions are very similar to the recursive functions allowed here. The output expressions are programs in which recursion has been eliminated when possible, loops have been merged, and destructive assignment and shared data structures used when permitted. Thus, the input is usually an elegant, easily understood program, while the output is an efficiently implemented but equivalent program. The implications are obvious: A programmer could write a procedure in a highly structured, recursive language. This would allow the methods of this thesis to be used to establish the correctness of the procedure. Then Darlington's methods could be used to produce an efficiently coded version which is known to have the same properties.

BIBLIOGRAPHY

de Bakker, J. W., and Scott, D. 1969. "A Theory of Programs", Unpublished memo., Vienna.

Bledsoe, W. W. 1971. "Splitting and Reduction Heuristics in Automatic Theorem Proving", Artificial Intelligence, Vol. 2, pp. 55-77, North Holland Publishing Company, Amsterdam.

Boyer, R. S. 1973. "Pretty Print", Department of Computational Logic, Memo. 64, University of Edinburgh.

Boyer, R. S., and Moore, J S. 1972. "The Sharing of Structure in Theorem-Proving Programs", Machine Intelligence 7, pp. 101-116, (eds. Meltzer, B., and Michie, D.), Edinburgh University Press, Edinburgh.

Brotz, D., and Floyd, R. W. 1973. "Proving Theorems by Mathematical Induction", Stanford Computer Science Department, (to appear).

Burstall, R. M., 1972. "Some Techniques for Proving Correctness of Programs which Alter Data Structures", Machine Intelligence 7, pp. 23-50, (eds. Meltzer, B., and Michie, D.), Edinburgh University Press, Edinburgh.

Burstall, R. M. 1969. "Proving Properties of Programs by Structural Induction", Computer Journal, Vol. 12, pp. 41-48.

Burstall, R. M., Collins, J. S., and Popplestone, R. J. 1971. Programming in POP-2, Edinburgh University Press, Edinburgh.

Cooper, D. C. 1971. "Programs for Mechanical Program Verification", Machine Intelligence 6, pp. 43-59, (eds. Meltzer, B., and Michie, D.), Edinburgh University Press, Edinburgh.

Curry, H. B., and Feys, R. 1958. Combinatory Logic, North Holland Publishing Company, Amsterdam.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. 1972. Structural Programming, Academic Press, London.

Darlington, J. 1972. "A Semantic Approach to Automatic Program Improvement", Ph.D. thesis, University of Edinburgh.

Floyd, R. W. 1967. "Assigning Meanings to Programs", in Proceedings of a Symposium in Applied Mathematics, Vol. 19 -- Mathematical Aspects of Computer Science, pp. 19-32, (ed. Schwartz, J. T.), American Mathematical Society, Providence, Rhode Island.

Gerhart, S. L. 1972. "Verification of APL Programs", Ph.D. thesis, Carnegie-Mellon University.

Good, D. I. 1970. "Toward a Man-Machine System for Proving Program Correctness", Ph.D. thesis, University of Wisconsin.

Goodstein, R. L. 1957. Recursive Number Theory, North Holland Publishing Company, Amsterdam.

Hoffman, G. R., and Veenker, G. 1971. "The Unit-Clause Proof Procedure with Equality", Computing, Vol. 7, pp. 91-105.

Katz, S. M., and Manna, Z. 1973. "A Heuristic Approach to Program Verification", Proceeding of IJCAI 1973, (to appear).

King, J. C. 1969. "A Program Verifier", Ph.D. thesis, Carnegie-Mellon University.

Kowalski, R., and Kuehner, D. 1971. "Linear Resolution with Selection Function", Artificial Intelligence, Vol. 2, pp. 227-260.

Manna, Z. 1969. "The Correctness of Programs", J. Comp. and System Science, Vol. 3, No. 2, pp. 119-127.

Manna, Z., and Pnueli, A. 1970. "Formalization of Properties of Functional Programs", JACM, Vol. 17, No. 3, pp. 555-569.

Manna, Z., Ness, S., and Vuillemin, J. 1972. "Inductive Methods for Proving Properties of Programs", in Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, pp. 27-50.

McCarthy, J. 1963. "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems, pp. 33-70, (eds. Braffort, P., and Hirschberg, D.), North Holland Publishing Company, Amsterdam.

McCarthy, J., and Painter, J. A. 1967. "Correctness of a Compiler for Arithmetic Expressions", in Proceedings of a Symposium in Applied Mathematics,Vol. 19 -- Mathematical Aspects of Computer Science, pp. 33-41, (ed. Schwartz, J. T.), American Mathematical Society, Providence, Rhode Island.

McCarthy, J., et al. 1969. LISP 1.5 Programmers Manual, MIT Press, Cambridge, Massachusetts.

Milner, R. 1972. "Logic for Computable Functions: Description of a Machine Implementation", Artificial Intelligence Memo. 169, Stanford University.

Milner, R. 1972. "Implementation and Applications of Scott's Logic for Computable Functions", _Proceedings of an ACM Conference on Proving Assertions about Programs_, SIGPLAN Notices, Vol. 7, No. 1, pp. 1-6.

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

Naur, P. 1966. "Proof of Algorithms by General Snapshots", _BIT_, Vol. 6, pp. 310-316.

Park, D. 1969. "Fixpoint Induction and Proofs of Program Properties", _Machine Intelligence 5_, pp. 59-78, (eds. Meltzer, B., and Michie, D.), Edinburgh University Press, Edinburgh.

Robinson, J. A. 1971. "Computational Logic: The Unification Algorithm", _Machine Intelligence 6_, pp. 63-72, (eds. Meltzer, B., and Michie, D.), Edinburgh University Press, Edinburgh.

Scott, D. 1970. "Outline of a Mathematical Theory of Computation", Oxford University Computing Laboratory, Programming Research Group, Technical Monograph PRG-2 (November,1970).

Shoenfield, J. R. 1967. _Mathematical Logic_, Addison-Wesley Publishing Company, Reading, Massachusetts.

Topor, R. W., and Burstall, R. M. 1973. Private communication.

Wegbreit, B. 1973. "Heuristic Methods for Mechanically Deriving Inductive Assertions", _Proceedings of IJCAI 1973_, (to appear).