STRATEGIES FOR MECHANIZING STRUCTURAL INDUCTION

Raymond Aubin

School of Computer Science
and Artificial Intelligence

University of Edinburgh
Edinburgh, Scotland, U. K.[1]

Abstract

A theorem proving system has been programmed for automating mildly complex proofs by structural induction. One purpose was to prove properties of simple functional programs without loops or assignments. One can see the formal system as a generalization of number theory: the formal language is typed and the induction rule is valid for all types. Proofs are generated by working backward from the goal. The induction strategy splits into two parts: (1) the selection of induction variables, which is claimed to be linked to the useful generalization of terms to variables, and (2) the generation of induction subgoals, in particular, the selection and specialization of hypotheses. Other strategies include a fast simplification algorithm. The prover can cope with situations as complex as the definition and correctness proof of a simple compiling algorithm for expressions.

Descriptive Terms

Program proving, theorem proving, data type, structural induction, generalization, simplification.

Introduction

In general, proving properties of programs requires an inductive argument of one sort or other. Structural induction is used in this theorem proving system for automating mildly complex proofs about programs written in a simple programming language without loops or assignments.

Theorem provers using such a method were written by Brotz (1974) for number theory and Boyer and Moore (1975) for a theory of lists (see also Moore (1973) and Moore (1975)). The latter was applied to proving properties of programs written in a LISP-like language. The present system is an improvement over these previous works by its typed language and it more sophisticated use of induction.

[1] Present address: Department of Computer Science, Concordia University, 1455 ouest, boul. de Maisonneuve, Montreal, Quebec, H3C 1M8, Canada.

After an overview of the formal system and the search strategy, the paper explains how induction variables are selected, which includes generalization, and how induction subgoals are generated. Finally, other strategies are presented, including simplification. A detailed example and technical remarks constitute the appendices. Aubin (1974) describes the whole system in detail.

Formal System

The formal system can be thought of as a generalization of number theory, but with implicit outermost universal quantifiers only.

Every term, that is, every variable and function application, has a type. Types and constructor constants are hierarchically introduced. For example,

```
[true: | false:] -* bool
[zero: | succ:nat] •* nat
[nil: | cons:nat,list] •+ list
[atom:nat | consx:sexpr,sexpr] -► sexpr
[nulltree: | tip:nat|
       node: tree,nat,tree]    tree
```

Variables are simply declared. Finally, defined function constants are introduced by stages with the help of definitions by cases (Burstall 1969, Hoare 1976). Here are some concrete examples:

```
a=>b | bool <=
cases a [true <= b |
         false <= true]

a&b | bool <-
(a=>(bU=>false))=>false

m=n | bool <=
cases m
  [zero <= cases n [zero <= true |
                    succ(n) <= false] |
   succ(m) <= cases n [zero <= false |
                       succ(n <= m=n]]
```

They introduce the function constants of implication, conjunction, and equality for terms of type nat. The computer program uses another concrete representation for type and function definitions as can be seen in Appendix 1.

The inference rules are those of (1) truth,(2) specialization,(3) definition by k-recursion,(4) modus ponens, (5) substitutivity of equality, and (5) induction.

The domain of interpretation is a many-sorted word algebra generated by the empty set. A lexicographic ordering is defined over the domain so that the principle of structural induction holds in it. An interpretation is given for the language which leads to a proof of soundness and weak completeness. In particular, the meaning of a function constant defined by cases is a well-defined k-recursive function.

This primitive system is raised by introducing more connectives (or, not, cond) and by deriving some inference rules. Terms are put in normal form by means of rules inspired from Ketonen's dialect of Gentzen's sequent calculus (1955). A thorough description of the formal system can be found in Aubin (1976).

Search Strategy

Proving theorems can be seen as a game: the formal system sets the rules besides which we have a strategy of play. This strategy must meet three criteria: (1) it must follow the rules of the same (a question of soundness), (2) it must be a winning strategy (a question of completeness), and (3) it must use a tolerable amount of resources (a question of efficiency).

The search strategy of the present theorem prover works backward, reducing the original goal to subgoals, which are in turn reduced to further subgoals,etc. A solution is found when there is no more subgoals to achieve. A procedure to reduce a goal to subgoals is called a tactic (Gordon, Milner, and Wadsworth 1977).

A necessary and sufficient condition of soundness of this strategy is that it never reduces a nonachievable goal to only achievable subgoals. This is fulfilled if the tactics are inverses of valid inference rules, primitive or derived. In particular, the tactic corresponding to the rule of truth reduces the goal true() to no further subgoals. Soundness means that when a solution is found, a proof is indeed found.

A necessary condition of completeness can be seen as the converse of the previous condition: an achievable goal must not be reduced to nonachievable subgoals. This is always fulfilled by tactics corresponding to rules which are actually invertible, either in general or in some context. As for tactics not bound to invertible rules, the theorem prover tries to find counter examples to the subgoals they generate: if it succeeds, the condition is not met and the subgoals, rejected; if it fails, we cannot tell for sure that the condition is satisfied, but we have some ground to believe that it is, and the subgoals are retained.

Finding sufficient conditions for completeness is the main problem of theorem proving and the remaining sections of this paper will describe my contribution in this direction.

The preceding points lead to considering the utilization of resources. A source of efficiency in the present prover is the fact that no backtracking takes place, that is, at each stage, only one way of reducing a goal is irretrievably taken. So, it is sufficient to keep a simple stack of goals : the original goal is pushed down onto it, each tactic reduces the goal on top, pops up the stack, and pushes down the new subgoals onto it. But above this structure, the choice and use of tactics have a greater bearing on efficiency. This prover uses the following tactics in turn, until the goal stack is empty:

(1) Simplification (inverse of a derived rule of substitutivity of equality and rule of definition by k-recursion)

(2) Splitting (inverse of a derived rule of conjunction: from t and s, infer t&s)

(3) Replacement (inverse of a derived rule of substitutivity) and strengthening (inverse of a derived rule of weakening: from t, infer s=t)

(4) Contraction (inverse of a derived rule of substitutivity)

(5) Truth (inverse of the rule of truth)

(6) Generalization (inverse of the specialization rule), induction (inverse of the induction rule), and strengthening.

The search is aborted if the current goal cannot be reduced by any tactic, e.g., the goal false().

Induction Variables and Generalization

The induction tactic has been divided into two distinct parts: (1) the selection of a list of variables to induce upon and (2) the generation of the induction subgoals, given these induction variables. This section treats the first aspect. Actually, I submit that selection of induction variables and generalization are intrinsically linked together; so, both will be studied in this section.

Boyer and Moore (1975) first put in evidence the fact that only recursion variables were suitable candidates as induction variables. I will further constrain their fundamental idea by focusing on certain recursion terms of particular importance. (A recursion term is a term which occurs in the argument position of a case variable.) If we allow ourselves to talk of (symbolic) evaluation regarding the application of k-recursive definitions, we may as well talk of computation rule. A computation rule tells us which subterm of a term to apply the k-recursion definition rule to. Nothing can be gained from a completeness point of view by introducing this notion, but it can lead to improved efficiency.

The call-by-need computation rule is known to be optimal for recursion equations (Vuillemin 1973) and can usefully be applied to our problem. The starting point is quite simple. What do we need to know about a function application in order to be able to apply the k-recursive definition rule to it? We need to know the values of its recursion terms, if it has any. The interesting point is that if we apply the call-by-need line of reasoning to an induction goal which has already been simplified, the process will be stopped by one or more variables marking argument positions

which the call-by-need evaluator must have more information about: I submit that these variable occurrences constitute excellent candidates for doing induction upon. I call them primary variable occurrences.

A simple example can helpfully illustrate this. Take the goal:

```
(j<>k)<>l=j<>(k<>l).
```

The infix function constant <> denotes the function which appends two lists and is defined thus:

```
k<>l | list <=
cases k [nil <= l |
         cons(n,k) <= cons(n,k<>l)].
```

We start the chain of reasoning with the function constant -; both of its arguments are recursion arguments. So, we need to evaluate both of them before trying to apply the definition of =. We iterate the process: to know about **(j<>k)<>l,** we must know about **j<>k,** and to know about **j<>k,** we must know about **j.** But we know nothing about j; so, this primary occurrence of j makes a good induction candidate. On the right of the equality, to evaluate j<>(k<>l), we must know about j again. So, this variable is undoubtedly the induction variable to choose according to this technique. Note its directedness: k and 1 are never considered. And indeed, this theorem is proved automatically in one induction on (j).

The efficiency of this approach is put in evidence if we replace j by cons(n,j) as would be done in the generation of an induction conclusion. Primary variable occurrences are the only ones which, once replaced by structures, allow evaluation to be eventually applicable to the whole goal (try with k and 1).

The interesting fact about this approach to induction variable selection is that generalization can be integrated to it in a natural way. Which term occurrences in the goal can we consider as better candidates for induction than the primary variables? The answer is simple: the term occurrences leading to them by the call-by-need evaluation, or in other words, the term occurrences in which the primary variable occurrences appear. I call these primary term occurrences, including the primary variable occurrences.

The strong relation between selection of induction variables and generalization is theoretically supported by Prawitz's results (1971).

Here is an example with the same flavour as the previous one (the function constant rev denotes the reverse function on lists):

```
(rev(j)<>k)<>l=rev(j)<>(k<>l).
```

We do as before except that for each term occurrence considered by the call-by-need evaluator, we ask the question: can this occurrence (may be together with others) be generalized? This is answered negatively or positively according to whether the prover can or cannot find a counter-example to the generalized subgoal. In this example, the answers are negative until we get to rev(j): if we replace both occurrences of it by a variable, the new subgoal is still achievable (it is actually the same as in the previous example). The new variable is chosen to be the induction variable.

The advantage of this purposeful generalization is that we can meaningfully generalize only certain occurrences of a term and in particular of a variable. For example, with:

```
(j<>j)<>j=j<>(j<>j),
```

we find that the first and the fourth occurrences of j are primary occurrences. We try to generalize them to a new variable which successfully yields:

```
(k<>j)<>j=k<>(j<>j)
```

This subgoal can be proved in one induction on (k).

Note two points: (1) a search mechanism for counter-examples is essential to such a generalization method and (2) the approach of Brotz (1974), and Boyer and Moore (1975) to generalization as separated from induction variable selection leads in this example to the nonachievable subgoal **k<>j=j<>k.**

Some pragmatic aspects must be taken into account in the implementation of this method. In particular, since searching for counter-examples is time-consuming, we limit generalization to the cases which have a better chance of success, i.e., when the term occurrences to generalize appear on both sides of an equality or implication (Boyer and Moore 1975, Brotz 1974). In addition, the above method may propose several candidates and the system uses some tie-breaking rules to elect a unique one.

Here is an additional example of generalization. The original goal is:.

```
subset (k,k).
```

The function constant subset is defined by cases on its first argument. No generalization is possible, in the goal and induction is done on (k). We obtain an induction subgoal for which the induction hypothesis cannot be used:

```
subset(k[1],k[1])
=>
subset(k[1],cons(n[1],k[1])).
```

The first and third occurrences of k[l] are primary and can now be generalized to yield the subgoal:

Subset (k[2],[k]1)
■>
Subset (k[2], cons(n[1], k[1])),
which is easily proved in one induction on (k[2]).

Induction Subgoal Generation

We now want to find the induction subgoals, given the list of induction variables. In particular, we need to find heuristically justified instantiations for the induction hypotheses. We may also wish to discard some hypotheses judged useless; it should be clear that this can cut down the complexity of the subgoals considerably.

In order to generate the induction subgoals, Boyer and Moore (1975) use a method which maps the structure of what they call a bomb list into the required terms. The bomb list of a goal contains information about how definitions fail to apply to the goal. In Moore's later version (1975), the corresponding mechanism is directly based on function definitions. I would argue that such techniques whereby induction subgoals are more or less directly constructed from function definitions do not constitute a sound approach. In particular, the soundness of the Boyer-Moore tactic is not provable, at least for their system.

In my tactic, the heuristic-part is separated from the nonheuristic part. On the one hand, all induction subgoals are generated on the basis of type definitions. For each of them, the conclusion and all the hypotheses are considered. Since checking the admissibility of type definitions is straightforward, it is easy to convince oneself of the soundness of this nonheuristic part.

On the other hand, the role of the definitions of the function constants appearing in the induction goal does not so beyond giving information about the rejection, or the acceptance and instantiation of tentative induction hypotheses, i.e. about the heuristic part. Now, discarding an induction hypothesis from an induction subgoal is sound (by the weakening rule) and preserves the achievability of the induction subgoal. Moreover, instantiating an induction hypothesis is justified by the induction rule.

Induction conclusions and hypotheses are actually represented as substitutions involving the induction variables. By applying these substitutions to the induction goal and bundling up the resulting terms with ■> and &, we easily obtain the induction subgoals themselves.

The induction tactic first finds the conclusion substitutions. For each variable, the algorithm generates the structures representing all the values that can be assumed by the variable (the system can do induction from any number of bases). Then, the conclusion substitutions are constructed by successively binding the induction variables to each of their corresponding structures.

As an example, take the induction goal ack(n,m) >zero; the induction variables (n,m) are selected. The function ack is defined thus:

```
ack(n,m) | nat <=
cases n
[zero <= succ(m) |
 succ(n) <=
cases m
[zero <= ack(n,succ(zero)) |
 succ(m) <= ack(n,ack(succ(n),m))]].
```

We set as possible substitutions for n, the closed structure zero and the open structure, say, succ(n[1]); similarly, zero and succ(m[1]) are associated with m. This means that there are four conclusion substitutions: (1) [zero/n] [zero/m], (2) [zero/n] [succ(m[1])/m], (3) [succ(n[1])/n] [zero/m], and (4)[succ(n[1])/n] [succ(m[1])/m].

Next, for each conclusion substitution, we have to find zero or more hypothesis substitutions, according to our lexicographic ordering.

Consider any conclusion substitution. We simply have to find the immediate predecessors of the list c*(x*) of structures bound to the induction variables, that is, for all i(l<i<n), **the list c[1](x[1]*),..., c[i-1](x[i-1]*), x[i,j], s[i+1], ..., s[n], where x[i,j] in x[i]* has the same type as c[i](x[i]*) and s[j] (i+1<j<n) is any term.**

In our examples, we get the following results:

(1) No substitutions, since zero has no proper substructures.

(2) [zero/n] [m[1]/m]

(3) [n[1]/n] [s[1]/m], where s[1] is any term

(4) [n[1]/n] [s[2]/m], where s[2] is any term, and [succ(n[1])/n] [m[1]/m].

Function definitions come into the picture to serve two purposes: (1) to reject a hypothesis substitution if no use can be foreseen for it, and (2) to find relevant instances for the free variables. Roughly speaking, the strategy applies hypothesis and conclusion substitutions to the induction goal, simplifies the resulting terms (in particular, using function definitions), and then tries to match parts of these terms: a failure counts toward rejection of the hypothesis, while a success both counts toward its retention and provides instances for the free variables.

In the Ackermann's example, we have that:

(1) There is already no hypotheses

(2) The tentative hypothesis is discarded

since the definition of ack is not recursive for this case and matching cannot even be attempted.

(3) By applying the definition of ack to the conclusion and matching, we find the instance succ(zero) for the free variable »[i].

(4) There are two recursive calls of ack for this case; we set two matches and retain both hypotheses, letting s[2] be ack(succ(n[l]),m[l]).

So, finally, the four following induction subgoals are generated:

(1) ack(zero,zero) > zero

(2) ack(zero,succ(m[l])) > zero

(3) ack(n[l],succ(zero)) > zero
=> ack(succ(n[l]),zero) > zero

(4) **ack(n[1],ack(succ(n[1]),m[1])) > zero**
& ack(succ(n[1]),m[1]) > zero
=> ack(succ(n[1]),succ(m[1])) > zero

This method is not fool proof: it will sometimes retain hypotheses which are in fact useless (as above), and sometimes discard useful hypotheses. But, in general, it errs on the safe side.

Other Strategies

Other strategies are not so much directly related to using the induction rule.

Indirect Generalization

In the following definition:

```
rev2a(l,k) | list <=
cases l [nil <= k |
         cons(n,l) <= rev2a(l,cons(n,k))],
```

the nonrecursion argument k does not stay fixed on the right, but becomes cons(n,k). The interest of such definitions lies in the fact that for the class of problems studied, they are literal translations of iterative programs. Such nonfixed nonrecursion arguments are called accumulators (following Moore (1975)), since they can be considered as holding current values of computations.

Quite often, accumulators have to be generalized when they are not variables. For example, we should generalize nil in the goal rev2a(k,nil) -rev(k). Since it will not match cons(n[l],nil) in the simplified conclusion of an induction on (k). However, we do not have an occurrence on both sides of «. How can we massage our goal so as to make nil recur on the right of the equality? Intuitively, if we know that l<>nil"l, we can rewrite rev(k) as rev(k)<>nil. So, the goal becomes rev2a(k,nil)«rev(k)<>nil, and nil occurs on both sides. What if we replace it by a new variable? We get rev2a(k,l)=rev(k)<>l, which is proved easily by inducing on (k), since 1 can now be replaced by cons(n[l],l) in the induction hypothesis. Similar generalizations can be found automatically for natural numbers and lists by a method using specialization as a means of achieving generalization.

Replacement and Strengthening

These tactics are responsible for using the induction hypotheses and is an adaptation of a method already experimented with by Brotz (1974) and especially Boyer and Moore (1975). For those members of the antecedent of a goal which are equalities, it tries to replace the right by the left-hand side, or vice-versa, in one or more members of the consequent. So, grossly speaking, it reduces s=t=>u[t/z] to s=t=>u[s/z], or vice versa. A strengthening tactic is used concurrently. In effect, the antecedent members of an implication which are involved in replacement are discarded from the antecedent, i.e. s=t«>u[s/z] is reduced to u[s/z]. This is called strenghening since it is the inverse of the weakening rule.

These tactics are well justified and preserve achievability when they involve induction hypotheses, but replacement with equalities not constrained by induction requires a new approach.

Splitting

The prover splits conjunctions, that is, it reduces a goal of the form s&t to the subgoals s and t. This tactic preserves achievability. Brotz (1974) used it, but not Boyer and Moore (1975).

Contraction

This tactic reduces a goal of the form f(s[1],...,s[n]) = f(t[1],...,t[n]) to s[i]=t[i], where s[j] is identical to t[j] (1≤j≤i-1 , i+1≤j≤n) and s[i] differs from t [i]. This is actually applied to any consequent member of a term in normal form. The tactic is justified by the substitutivity of equality; however, it does not preserve achievability. A similar strategy can be found in Brotz (1974), but not in Boyer and Moore (1975).

Simplification

This is the most important tactic besides induction. The simplification problem splits into three subproblems: (1) one of logical equivalence between terms before and after simplification, (2) one of complexity measure for terms, and (3) one of selection,i.e., what to replace by what in the terms to be simplified. This last question is perhaps the most interesting.

The method used in this prover is inspired from Vuillemin's call-by-need computation rule (1973). Applied to simplification, the rule says: select the leftmost-outermost subterm which can be simplified (i.e. cal1-by-name), but take

the maximum advantage of shared subterms. Because all terms have the same internal representation (a three-field record), the tactic can deal with variables and function applications indistinguishably; moreover the program which applies a substitution does not do undue copying. So, once a term t has been fully simplified, the resulting term s, whether it is a variable or not, is copied into the record of term t, whose boolean field is set to true. Thus any superterm which shared term t now shares its simplified equivalent s.

The second half of the selection question concerns the order in which the various simplification rules are applied on a given term. This prover tries successively (1) pure simplification rules, (2) k-recursive definitions, and (3) normalization rules. The rules are further ordered within each category according to other criteria.

The gain in efficiency due especially to the sharing of structures is very important.

Conclusion

The strong points of this theorem proving system are (1) its typed language, (2) its mechanism for selecting induction variables and generalizing, (3) its sound way of generating induction subgoals, and (4) its fast simplification algorithm.

However, its formal system is still too weak: one would like to relax the restrictions on quantification, and on type and function definitions. It is also clear that the pure backward search is too limiting and the discovery of useful lemmas on a reasonably large scale will require more of the user (interactively or not).

Two recent works have proposed solutions to such problems and others addressed in this paper. Cartwright's system (1976) includes axiomatically defined types constrained to denote the same domains as the type constants described in this paper. Since his language is not typed (despite his claim), his induction rule is more powerful. Moreover, his system accepts all computable functions. His search strategy is crucially dependent on interaction with a user.

Boyer and Moore (1977) have a new and even more powerful system since their types can be defined axiomatically in total freedom. It can accept any total function (then one can only conclude that their language is not enumerable:). The additional typing freedom increases the complexity of the proofs found by their automatic search strategy, but they have the facility of specifying any useful lemmas to the prover prior to attempting a proof.

Acknowledgements

I am most grateful to my directors of studies, Robin Milner and Rod Burstall, and also to Bob Boyer and J Moore. I was supported by the Commonwealth Scholarship Commission and the Conseil national de recherches du Canada.

Appendix 1

Compiling Algorithm for Expressions

In the concrete syntax used by the computer program, type and function definitions are input as POP-2 lists. The following simple compiling algorithm for expressions illustrates the use of such definitions. Similar algorithms can be found in Burstall (1969); in Milner and Weyrauch (1972) and Cartwright (1976), who obtained a correctness proof interactively by machine; and in Boyer and Moore (1977) who got an automatic proof as I did. Note the presence of vacuously defined type and function constants.

We start by defining the syntax of the source language of expressions by means of type definitions:

```
[NAME]
[OPERATOR]
CEXPRESS [SIMPLE NAME]
         [COMPOUND OPERATOR EXPRESS EXPRESS]]
```

Written in the form used in the body of this paper, this last type definition, for example, would read; [simple:name | compound:operator,express, express] -► express.

Type definitions are also used for the semantic domains. States are intended to map names to numbers. Our first-order logic forces us to give a function which applies an object of type FUNCTION to two numbers. We assume that the variables F, and M and N have been declared to be of type FUNCTION and NAT respectively:

```
[FUNCTION]
[NAT [ZERO] [SOCC NAT]]
[STATE]
[[[APPLY F M N] NAT] []]
```

The following semantic functions give the meaning of the syntactic constructs; MSE can be said to be an interpreter. NM is a variable of type NAME; ST, of type STATE; OP, of type OPERATOR; and E, E1, and E2, of type EXPRESS:

```
[[[LOOKUP NM ST] NAT] []]
[[[MO OP] FUNCTION] []]
[[[MSE E ST] NAT]
[CASES E
      [[SIMPLE NM] [LOOKUP NM ST]]
      [[COMPOUND OP E1 E2]
       [APPLY [MO OP] [MSE E1 ST] [MSE E2 ST]]]]]
```

Next, v/e turn to the taraet lanouaae. Syntactically, it is a set of programs which are lists of instructions. Postfixed notation is used. We also give a function to concatenate two target language programs. PR, PRI, and PR2 are variables of type PROGRAM; and IN, of type INSTRUCT:

```
[INSTRUCT [OPERATE OPERATOK] [FETCH NAME]]
[PROGRAM [NULLPR] [ADD INSTRUCT PROGRAM]]
[[[CONCAT PRI PR2] PROGRAM]
 [CASKS PRI
        [[NULLPR] PR2]
        [[ADD IN PRI]
         [ADD IN [CONCAT PRI PR2]]]]]
```

We define the semantic domains for the target language (pushdowns and stores), together with selecting functions for inspecting their constituents. PD is a variable of type PUSHDOWN; and STR, of type STORE:

```
[PUSHDOWN [EMPTY] [PUSH NAT PUSHDOWN]]
[STORE [MKSTORE STATE PUSHDOWN]]
[[[TOP PD] NAT]]
 [CASES PD
        [[EMPTY] [ZERO]]
        [[PUSH N PD] N]]]
[[[POP PD] PUSHDOWN]
 [CASES PD
        [[EMPTY] [EMPTY]]
        [[PUSH N PD] PD]]]
[[[STOF STR] STATE]
 [CASES STR [[MKSTORE ST PD] ST]]]
[[[PDOF STR] PUSHDOWN]
 [CASES STR [[MKSTORE ST PD] PD]]]
```

We have two semantic functions for the target language; they can be said to execute programs:

```
[[[DO IN STR] STORE]
 [CASES IN
        [[FETCH NM]
         [MKSTORE [STOF STR]
                  [PUSH [LOOKUP NM [STOF STR]]
                        [PDOF STR]]]]
        [[OPERATE OP]
         [MKSTORE [STOF STR]
                  [PUSH [APPLY [MO OP]
                               [TOP [POP [PDOF STR]]]
                               [TOP [PDOF STR]]]
                        [POP [POP [PDOF STR]]]]]]]]
[[[MT PR STR] STORE]
 [CASES PR
        [[NULLPR] STR]
        [[ADD IN PR] [MT PR [DO IN STR]]]]]
```

Finally, the function COMP compiles an expression, that is, it translates it into a program:

```
[[[COMP E] PROGRAM]
 [CASES E
        [[SIMPLE NM] [ADD [FETCH NM] [NULLPR]]]
        [[COMPOUND OP E1 E2]
         [CONCAT [COMP E1]
                 [CONCAT [COMP E2]
                         [ADD [OPERATE OP] [NULLPR]]]]]]]
```

The statement of correctness of this algorithm is:

```
[EQST [MT [COMP E] STR]
      [MKSTORE [STOF STR]
               [PUSH [MSE E [STOF STR]] [PDOF STR]]]]
```

In other words, we get the same store if we compile an expression and execute the resulting program, given a storage, as if we interpret the expression with the state of the given store and push the result down onto the stack of the store, leaving its state unchanged.

This statement can be proved automatically by the theorem prover with the help of the lemma:

```
[EQST [MT [CONCAT PR1 PR2] STR]
      [MT PR2 [MT PR1 STR]]]
```

which can be proved automatically on its own.

Appendix 2

Note on Implementation and Results

The prover is implemented in POP-2. The time taken for finding a proof varies from a few seconds to a few hundred seconds. This is essentially dependent on the extent in which counter-examples have to be searched for. The proof of the compiling algorithm, which does not involve any generalizations, takes only 25 seconds. This prover could prove most theorems in Brotz (1974) and Boyer and Moore (1975), plus many new ones. These theorems were proved using only a core of basic lemmas in the subtheory of booleans. However, the proofs of some of the hardest theorems, (e.g., the compiler correctness) required that equalities of other subtheories be added to the set of simplification rules before they were attempted.

Bibliography

Aubin, Raymond. "Mechanizing structural induction." Ph.D. thesis, University of Edinburgh, Edinburgh, 1976.

Boyer, Robert S., and Moore, J. "A lemma driven automatic theorem prover for recursive function theory." Computer Science Lab. SRI, 1977.

Boyer, Robert S., and Moore,J. "Proving theorems about LISP functions." J. ACM 22, 1 (1975): 129-144.

Brotz, Douglas K. "Embedding heuristic problem solving methods in a mechanical theorem prover." STAN-CS-74-443, Computer Science Dept., Stanford Univ. 1974.

Burstall, R.M. "Proving properties of programs by structural induction." Comp.J. 12, 1 (1969): 41-48.

Cartwright, Robert. "User-defined data types as an aid to verifying LISP programs." In Proc. Coll. Aut. Lang, and Prog., pp.228-256. Ed. S. Michaelson and R. Milner. Edinburgh: Edinburgh University Press, 1976.

Gentzen, Gerhard. Recherche sur la deduction logique. Trad, et coram. R. Feys et J. Ladriere. Paris: PUF, 1955.

Gordon, M.; Milner, R.; and Wadsworth, C. "Edinburgh LCF." Dept. of Computer Science, Univ. of Edinburgh, 1977.

Hoare, C.A.R. "Recursive data structures." Int. J. Comp. and Inf. Sc. 4,2 (1975): 105-132.

Milner, R., and Weyrauch, R. "Proving compiler correctness in a mechanized logic." In Machine Intelligence 7, pp. 51-70. Ed. B.Meltzer and D. Michie. Edinburgh: Edinburgh University Press, 1972.

Moore, J Strother. "Computational logic: structure sharing and proof of program properties." Ph.D. thesis, Univ. of Edinburgh, Edinburgh, 1973.

Moore, J Strother. "Introducing iteration into the pure LISP theorem-prover." IEEE Trans. Soft. Eng. 1, 3 (1975): 328-338.

Prawitz, Dag. "Ideas and results of proof theory." In Proc. 2nd Scand. Log. Symp., pp. 235-307. Ed. J. Fenstad. Amsterdam: North-Holland, 1971.

Vuillemin, Jean E. "Proof techniques for recursive programs." Memo AIM-218/STAN-CS-73-393, Computer Science Dept., Standord Univ. 1973.