MECHANIZING STRUCTURAL INDUCTION
PART II: STRATEGIES*

Raymond AUBIN

Department of Computer Science, Concordia University, Montréal, Québec H3C 1M8, Canada**

Communicated by M. Nivat
Received November 1977
Revised March 1978

Abstract. A theorem-proving system has been programmed for automating mildly complex proofs by structural induction. 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 relevant 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.

1. 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 recursive functions.

Theorem provers using such a method were written by Brotz [7] for number theory and Boyer and Moore [6] for a theory of lists (see also Moore [15, 16]). The latter was applied to proving properties of programs written in a LISP-like language. The present system is an improvement over previous works mainly by its typed language and its more sophisticated use of induction.

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 [1] describes the whole system in detail.

* A version of this paper was presented at the 5th International Joint Conference on Artificial Intelligence, 1977, Cambridge, MA.
** Present address: Bell-Northern, Ottawa, Ontario, K1Y 4H7, Canada.
2. 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,

\[
\begin{align*}
\text{true:} & \rightarrow \text{bool} \\
\text{false:} & \rightarrow \text{bool} \\
\text{zero:} & \rightarrow \text{nat} \\
\text{succ:} & \rightarrow \text{nat} \\
nil: & \rightarrow \text{list} \\
\text{cons:} & \rightarrow \text{nat, list} \\
\text{atom:} & \rightarrow \text{sexpr} \\
\text{nulltree:} & \rightarrow \text{tree}
\end{align*}
\]

Variables are simply declared. Finally, defined function constants are introduced by stages with the help of definitions by cases [8, 13]. Here are some concrete examples:

\[
\begin{align*}
a \Rightarrow b : \text{bool} & \Leftarrow \\
\text{cases } a & \left[ \text{true } \Leftarrow b \right. \\
& \left. \text{false } \Leftarrow \text{true} \right]\end{align*}
\]

\[
\begin{align*}
a \& b : \text{bool} & \Leftarrow \\
(a \Rightarrow (b \Rightarrow \text{false})) & \Rightarrow \text{false}
\end{align*}
\]

\[
\begin{align*}
m = n : \text{bool} & \Leftarrow \\
\text{cases } m & \left[ \text{zero } \Leftarrow \text{cases } n \left[ \text{zero } \Leftarrow \text{true} \right. \\
& \left. \text{succ}(n_1) \Leftarrow \text{false} \right]\right] \\
& \text{succ}(m_1) \Leftarrow \text{cases } n \left[ \text{zero } \Leftarrow \text{false} \right. \\
& \left. \text{succ}(n_1) \Leftarrow m_1 = n_1 \right]
\end{align*}
\]

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 (6) 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 derivir., some inference rules. Terms are put in normal form by means of rules inspired from Ketonen's dialect of Gentzen's sequent calculus [11]. A thorough description of the formal system can be found in Aubin [1, 2].

3. 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 game (a question of soundness),
2. it should 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 are no more subgoals to achieve. A procedure to reduce a goal to subgoals is called a tactic [12].

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 [1, 2] 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. In other words, only refutations are decisive.

Finding sufficient conditions for completeness is the main problem of theorem proving and the remaining sections of this paper will describe my contributions 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 \( t \land s \)),
(3) replacement (inverse of a derived rule of substitutivity) and strengthening
(inverse of a derived rule of weakening: from \( t \), infer \( s \Rightarrow t \)),
(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 \( \text{false} \).

4. 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.

At the basis of the method for selecting induction variables is the following
observation: proving a term by induction is a finite way of expressing the infinite
number of computations in which the term is evaluated to true ( ) for all values of its
variables. Using a similar analogy, Boyer and Moore [6] 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
[18] 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:

\[(j\langle k \rangle \langle l \rangle) = j\langle k \rangle \langle l \rangle.\]

The infix function constant \(\langle \rangle\) denotes the function which appends two lists and is defined thus:

\[
k\langle \rangle l : \text{list} \leftarrow
\begin{cases}
\text{nil} \leftarrow l \\
\text{cons}(n, k\langle l \rangle) \leftarrow \text{cons}(n, k\langle l \rangle). 
\end{cases}
\]

We start the chain of reasoning with the function constant \(\langle \rangle\); 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 \((j\langle k \rangle \langle l \rangle)\), we must know about \(j\langle k \rangle\), and to know about \(j\langle k \rangle\), 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 \(j\langle k \rangle \langle l \rangle\), 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 \(l\) 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 \(\text{cons}(n, j_1)\) 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 \(l\)).

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 [17].

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

\[(\text{rev}(j)\langle k \rangle \langle l \rangle) = \text{rev}(j)\langle k \rangle \langle l \rangle.\]

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 refute the generalized subgoal. In this example, the answers are negative until we get to \(\text{rev}(j)\): 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:

\[(j \circ j) \circ j = j \circ (j \circ j),\]

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:

\[(k \circ j) \circ j = k \circ (j \circ j).\]

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 [7] and Boyer and Moore [6] to generalization as separated from induction variable selection leads in this example to the nonachievable subgoal \( k \circ j = j \circ k. \)

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 (see [6, 7]). 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:

\[ \text{subset}(k, k) \].

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:

\[ \text{subset}(k_1, k_1) \Rightarrow \text{subset}(k_1, \text{cons}(n_1, k_1)). \]

The first and third occurrences of \( k_1 \) are primary and can now be generalized to yield the subgoal:

\[ \text{subset}(k_2, k_1) \Rightarrow \text{subset}(k_2, \text{cons}(n_1, k_1)). \]

which is easily proved in one induction on \((k_2)\).
5. 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 [6] 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 [16], the corresponding mechanism is directly based on function definitions.

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 go 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 \( \Rightarrow \) 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 \( \text{ack}(n, m) > \text{zero} \); the induction variables \((n, m)\) are selected. The function \( \text{ack} \) is defined thus:

\[
\text{ack}(n, m) : \text{nat} \leftarrow \\
\text{cases } n \begin{cases} 
\text{zero} & \leftarrow \text{succ}(m) \\
\text{succ}(n_1) & \leftarrow \\
\text{cases } m \begin{cases} 
\text{zero} & \leftarrow \text{ack}(n_1, \text{succ}(\text{zero})) \\
\text{succ}(m_1) & \leftarrow \text{ack}(n_1, \text{ack}(\text{succ}(n_1), n_1)) \end{cases} 
\end{cases}
\]

We get as possible substitutions for \( n \), the closed structure zero and the open structure, say, \( \text{succ}(n) \); similarly, zero and \( \text{succ}(m) \) are associated with \( m \). This means that there are four conclusion substitutions:

1. \([\text{zero}/n][\text{zero}/m]\),
2. \([\text{zero}/n][\text{succ}(m)/m]\),
3. \([\text{succ}(n)/n][\text{zero}/m]\) and
4. \([\text{succ}(n)/n][\text{succ}(m)/m]\).

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 of structures bound to the induction variables [1, 2].

In our example, we get the following results:

1. no substitutions, since zero has no proper substructures,
2. \([\text{zero}/n][\text{zero}/m]\),
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 \([\text{succ}(n)/n][\text{zero}/m]\).

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, 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 \( \text{ack} \) is not recursive for this case and matching cannot even be attempted,
3. by applying the definition of \( \text{ack} \) to the conclusion and matching, we find the instance \( \text{succ}(\text{zero}) \) for the free variable \( s_1 \),
4. there are two recursive calls of \( \text{ack} \) for this case: we get two matches and retain both hypotheses, letting \( s_2 \) be \( \text{ack}(\text{succ}(n_1), m_1) \).

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

1. \( \text{ack}(\text{zero}, \text{zero}) \text{> zero.} \)
2. \( \text{ack}(\text{zero}, \text{succ}(m_1)) \text{> zero,} \)
3. \( \text{ack}(n_1, \text{succ}(\text{zero})) \text{> zero} \)
   \( \Rightarrow \text{ack}(\text{succ}(n_1), \text{zero}) \text{> zero,} \)
4. \( \text{ack}(n_1, \text{ack}(\text{succ}(n_1), m_1)) \text{> zero} \)
   \( \& \text{ack}(\text{succ}(n_1), m_1) \text{> zero} \)
   \( \Rightarrow \text{ack}(\text{succ}(n_1), \text{succ}(m_1)) \text{> zero.} \)
This method is not foolproof: 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.

6. Other strategies

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

6.1. Indirect generalization

In the following definition:

```
rev2a(l, k) : list <=
cases l [nil <= k |
    cons(n, l) <= rev2a(l1, cons(n, k))]
```

the nonrecursion argument \( k \) does not stay fixed on the right, but becomes \( \text{cons}(n, k) \). The interest of such definitions lies in the fact that for the class of problems studied, they are literal translations of iterative programs. Such non-fixed non-recursion arguments are called accumulators (following Moore [16]), 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 \( \text{nil} \) in the goal \( \text{rev2a}(k, \text{nil}) = \text{rev}(k) \), since it will not match \( \text{cons}(n_1, \text{nil}) \) 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 \( \text{nil} \) recur on the right of the equality? Intuitively, if we know that \( L\text{nil} = l \), we can rewrite \( \text{rev}(k) \) as \( \text{rev}(k)\text{nil} \). So, the goal becomes \( \text{rev2a}(k, \text{nil}) = \text{rev}(k)\text{nil} \), and \( \text{nil} \) occurs on both sides. What if we replace it by a new variable? We get \( \text{rev2a}(k, l) = \text{rev}(k)\text{l} \), which is proved easily by inducing on \( k \), since \( l \) can now be replaced by \( \text{cons}(n_1, l) \) 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.

6.2. Replacement and strengthening

These tactics are responsible for using the induction hypotheses and is an adaptation of a method already experimented with by Brotz [7] and especially Boyer and Moore [6]. For those members of the antecedent of a goal which are equalisti
c, 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 \( s = t \Rightarrow u[t/z] \) to \( s = t \Rightarrow u[s/z] \), 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., $s = t \Rightarrow u[s/z]$ is reduced to $u[s/z]$. This is called strengthening 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.

6.3. 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 [7] and Boyer and Moore [6] use it.

6.4. Contraction

This tactic reduces a goal of the form $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ to $s_i = t_i$, where $s_i$ is identical to $t_i$ (for $1 \leq j \leq i - 1$, $i + 1 \leq j \leq n$) 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 [7] but not in Boyer and Moore [6].

6.5. 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 [18]. Applied to simplification, the rule says: select the leftmost-outermost subterm which can be simplified (i.e. call-by-name), but take the maximum advantage of shared subterms. Because all terms have the same internal representation, 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 in place 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.

7. 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 consistent 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 by Cartwright [9, 10] and Boyer and Moore themselves [5] have also had the goal of improving upon Boyer and Moore [6]. Their formal systems are discussed in Aubin [2]. From a search strategic point of view, these two systems leave more room to use guidance.

Cartwright's prover [9, 10] is completely interactive. This means that most tactics must be explicitly invoked by the user through various commands. Only the simplifier, which is built around user-supplied conditional rewrite rules, is called automatically.

Boyer and Moore [5] also make use of simplification based on user-supplied conditional rewrite rules, made available as previously proved theorems. However, they retain the automatic mode of their earlier version. Their strategy for instantiating induction hypotheses is quite similar to the one explained in this paper.

In summary, both Cartwright [9, 10] and Boyer and Moore [5] coupled more powerful formal systems with more user guidance which led to proving theorems of increased difficulty. However, they did not explore the generalization problem.

Acknowledgment

This work was carried out at the School of Computer Science and Artificial Intelligence, University of Edinburgh. I am especially 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 [8]; Milner and Weyrauch [14] and Cartwright [9, 10], who obtained a
correctness proof interactively by machine; and in Boyer and Moore [4] 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]
- [EXPRESS [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] \rightarrow 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] [SUCC 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, we turn to the target language. 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, PR1, and PR2 are variables of type PROGRAM; and IN, of type INSTRUCT:

\[
\begin{align*}
&\text{[INSTRUCT [OPERATE OPERATOR] [FETCH NAME]]} \\
&\text{[PROGRAM [NULLPR] [ADD INSTRUCT PROGRAM]]} \\
&\text{[[[CONCAT PR1 PR2] PROGRAM]}
\end{align*}
\]

\[
\begin{align*}
&\text{[CASES PR1} \\
&\quad [\text{[NULLPR] PR2}] \\
&\quad [\text{[ADD IN PR1]}
\end{align*}
\]

\[
\begin{align*}
&\text{[ADD IN [CONCAT PR1 PR2]]]]
\end{align*}
\]

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:

\[
\begin{align*}
&\text{[PUSHDOWN [EMPTY] [PUSH NAT PUSHDOWN]]} \\
&\text{[STORE [MKSTORE STATE PUSHOWN]]} \\
&\text{[[[TOP PD] NAT]}
\end{align*}
\]

\[
\begin{align*}
&\text{[CASES PD} \\
&\quad [\text{[EMPTY] [ZERO]}] \\
&\quad [\text{[PUSH N PD] N]])
\end{align*}
\]

\[
\begin{align*}
&\text{[[[POP PD] PUSHDOWN]}
\end{align*}
\]

\[
\begin{align*}
&\text{[CASES PD} \\
&\quad [\text{[EMPTY] [EMPTY]}] \\
&\quad [\text{[PUSH N PD] PD]])
\end{align*}
\]

\[
\begin{align*}
&\text{[[[STOF STR] STATE]}
\end{align*}
\]

\[
\begin{align*}
&\text{[CASES STR [MKSTORE ST PD] ST]]}
\end{align*}
\]

\[
\begin{align*}
&\text{[[[PDOF STR] PUSHDOWN]}
\end{align*}
\]

\[
\begin{align*}
&\text{[CASES STR [MKSTORE ST PD] PD]]}
\end{align*}
\]

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

\[
\begin{align*}
&\text{[[[DO IN STR] STORE]}
\end{align*}
\]

\[
\begin{align*}
&\text{[CASES IN}
\end{align*}
\]

\[
\begin{align*}
&\quad [\text{[FETCH NM]}
\end{align*}
\]

\[
\begin{align*}
&\quad [\text{MKSTORE [STOF STR]}
\end{align*}
\]

\[
\begin{align*}
&\quad [\text{PUSH [LOOKUP NM [STOF STR]]}
\end{align*}
\]

\[
\begin{align*}
&\quad [\text{PDOF STR]]]}
\end{align*}
\]
Finally, the function COMP compiles an expression, that is, it translates it into a program:

\[
[[[[\text{COMP } E] \text{ PROGRAM}]]
\]

The state of correctness of this algorithm is:

\[
[[[\text{EQST } [\text{MT } [\text{COMP } E] \text{ STR}]]]
\]

In other words, we get the same store if we compile an expression and execute the resulting program, given a store, 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:

\[
[[[\text{EQST } [\text{MT } [\text{CONCAT } PR1 \text{ PR2}] \text{ STR}]]]
\]

which can be proved automatically on its own.

Appendix 2. Note on implementation and results

The prover is implemented in POP-2. This programming language makes list processing easy and its general record facility allows an easy representation of terms.
The program text adds up to roughly 4500 lines of formatted and commented POP-2 code, or alternatively occupies about 175 blocks on disc. It runs on a DECsystem10 with a KA10 processor and the compiled version occupies 33K of core on top of the sharable 11K of the POP-2 system. Because of the compact representation of the search space, relatively little extra store is needed in the course of generating proofs, so that most of them can be carried out without exceeding 50K.

The time taken for finding a proof varies from a few seconds to a few hundred seconds. This is essentially dependent on the extent to which counter-examples have to be searched for: it is a rather time-consuming strategy. The proof of the compiling algorithm, which does not involve any generalizations, takes only 25 seconds. This prover could prove most of Brotz's and Boyer and Moore’s theorems. The missing ones are due to a less sophisticated system for lemma discovery for Brotz’s theorems, and to a less developed way of using hypotheses for Boyer and Moore’s. Moreover, some theorems could not be proved because searching for counter-examples was too long. On the other hand, the elaborate system of types and the generalization strategy made possible the proofs of many new theorems.

Most of 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 simplication rules before they were attempted.

References