THREE LECTURES ON THEOREM-PROVING AND PROGRAM VERIFICATION

J Strother Moore

Rapporteur: Mr. R. Millichamp

Abstract

I. Brief History of Mechanical Theorem-Proving

Can we build a machine to prove theorems? Man has been trying to construct such a machine since before the creation of the first digital computers. Indeed, mechanical theorem-proving has played a fundamental role in the history of computer science and artificial intelligence: work in metamathematics by Herbrand, Godel, Church, Turing and others laid much of the foundations of computer science; early work on theorem-proving and symbolic logic by Newell, Shaw, Simon, McCarthy and others helped create the field of artificial intelligence; the work of J.A. Robinson and others on resolution led to new ways to look at many artificial intelligence problems. The successes and failures of mechanical theorem-proving still influence artificial intelligence work today.

In this lecture the speaker will try to acquaint the audience with the past and present answers to the question "How can we build a machine to prove theorems?" In addition it will be shown how mechanical theorem-proving ideas have been applied to such topics as data base retrieval, problem solving, the design and implementation of non-deterministic programming languages, and the program correctness problem.

II A Tour through a Working Theorem-Prover

We will look at the operation and abilities of the theorem-prover being developed by Boyer and Moore at SRI International. The system deals with a quantifier free first-order logic based on recursive functions. The system provides the user with a schematic way of axiomatizing "new" inductively defined types, (e.g. integers and lists), a facility for recursively defining new concepts, and an automatic theorem-prover.

The theorem-prover contains a large amount of built-in knowledge about how to use definitions and previously proved theorems, common ways to generalise conjectures and, most importantly and uniquely, how and when to appeal to the principle of mathematical induction. In essence, the theorem-prover is a codification of the techniques humans use when presented with certain kinds of theorems arising frequently in computer science and discrete mathematics.

The lecture will consist primarily of a detailed look at how the system proves a particular theorem and why it chooses to make the "moves" it makes. The formal logic and proof techniques are completely described in A Computational Logic by Boyer and Moore (Academic Press, 1979).

III Program Verification

Today the most active application for mechanical theorem-proving is in the area of program reliability. Given a precise formal statement of what a program is supposed to do, it is relatively easy to transform the question "Is this program correct?" to the question "Are these formulae theorems?". The latter question can then be submitted to a mechanical theorem-prover.

There are a variety of ways to transform the former question into the latter. One way is to transform the program into a mathematical function from one machine state to another. Another way is to define an interpreter for the language. Still another way is to annotate the program with "input/output" assertions and "invariants" and to generate formulae that express the idea that the invariants and output specifications are true whenever they are encountered, provided the input specifications were true initially.

The lecture will illustrate all these techniques using a simple programming language. The lecture concludes with a discussion of the verification condition generator written by Boyer and Moore for a subset of both FORTRAN 66 and FORTRAN 77. We will discuss the verification conditions generated for a particular FORTRAN program and how these formulae are proved by the Boyer-Moore theorem-prover.

I HISTORY

How does one prove theorems? How can we build a machine to prove theorems?

Because mechanical theorem-proving has its roots in mathematics, and because mathematicians and philosophers have long asked the questions above, it is difficult to put a date on when mechanical theorem-proving was born. For example, the idea of mechanical proof, in the sense that we think of it today, would not have surprised Leibniz (1646-1716) who, on the one hand perfected and presented to the Royal Society, London, a mechanical binary adder (also capable of multiplication, division and square root computations) and on the other hand believed that all reasoning could be reduced to an "algebra of thought".

In the early 20th century formal axiomatic systems were developed. Such systems are characterized by a set of "well-formed formulae", a set of "axioms" and a set of "inference rules" with which one may deduce "theorems" from the axioms and previously deduced theorems. A "proof" of some formula p is just a finite sequence of formulae, the last of which is p and each of which is either an axiom or is derived from the preceding formulae by a rule of inference.

For example, here is a proof of the formula (-A v A) in the logic of Russell and Whitehead from Principia Mathematica.

Proof of (-A v A).

1.	(Q → R) → ((P v Q) → (P v R))	Axiom 4
2.	(Q → A) → ((-A v Q) → (-A v A))	Subst into 1
3.	(Q → A) → ((A → Q) → (-A v A))	Def of " "
4.	((A v A) → A) →((A → (A v A)) → (-A v A))	Subst into 3
5.	(P v P) →P	Axiom 1
6.	(A v A) →A	Subst into 5
7.	(A → (A v A)) →(-A v A)	M.P. 4 and 6
8.	Q →(P v Q)	Axiom 2
9.	A →(A v A)	Subst into 8
10.	(-A v A)	M.P. 7 and 9

It is easy to determine whether a sequence of formulae is a proof; theorem-proving is the art of discovering a proof -- if any -- for a given formula.

The 1920's and 1930's saw the careful study of formal axiomatic systems, primarily to clarify the then extensive debates between the various schools of thought on how the newly uncovered paradoxes in the foundations of mathematics might be remedied. Hilbert proposed to formalize classical mathematics (e.g., arithmetic) in logic and undertake the proof of its consistency via constructive means. Starting in the 1920's, this program was undertaken by Hilbert, Ackermann, von Neumann, Herbrand and others. Among the interesting results was Herbrand's Theorem (1930), which is a constructive version of a theorem proved earlier by Skolem [34] that suggests a mechanical means for finding a proof when it exists.

In 1931, Goedel showed that there exist formal sentences of arithmetic that are true (in the intended interpretation) but unprovable. Furthermore, he showed that if arithmetic is consistent then its consistency cannot be proved in arithmetic. In a certain sense, this undermined Hilbert's program. However, thanks in large part to Hilbert and his school, the formal study of formal proofs had been born.

During this same period, Church, Turing, and Goedel (the latter following a suggestion by Herbrand) developed what turned out to be the equivalent notions of lambda-definable, Turing computable, and general recursive functions. These developments led, in 1936 and 1937, to the demonstrations that there were no decision procedures for arithmetic or first-order predicate calculus. It is perhaps ironic that the concepts that eliminated the hope that perfect theorem-provers could be built simultaneously formed part of the theoretical foundations for the development of the device that makes imperfect theorem-provers realizable and perhaps practical.

Among the first heuristic mechanical theorem-provers physically realized was the Logic Theory Machine, programmed in the mid-50's by Newell, Shaw and Simon. The Logic Theory Machine constructed proofs in the propositional calculus using the axioms and rules of inference of Principia Mathematica. A succinct description of the Logic Theory Machine and its capabilities is provided in Computers and Thought [13].

The Logic Theory Machine attacked its problems in much the same way a human might attack them, when limited to the axioms, rules of inference and previously proved theorems of Principia Mathematica. The program contained four "methods" or "heuristics" for decomposing the given problem into "simpler" subproblems (e.g., instances of the axioms). An executive routine selected the methods to be tried and the subproblems to be worked on.

It should be observed that the authors of this early program were not so much concerned with answering the question "Is this propositional formula a theorem?" as they were the question "How does one go about solving hard problems?" Judged by its ability to answer the former question, the logic Theory Machine was not impressive. It was able to prove only 38 of the 52 propositional theorems in Chapter 2 of Principia. By contrast, Wang's algorithm [36], published in January, 1960 and based on the "semantic" idea of attempting to construct an assignment for falsifying a formula, was able to announce the validity of all of the propositional theorems in Principia.

But the Logic Theory Machine was a significant contribution to the infant field of "artificial intelligence" and was the first program to confront the hard problem that is unavoidable in the nonpropositional case: how does one choose which of many alternatives to pursue? The Logic Theory Machine inspired several other early AI programs, notably Gelernter's Geometry Theorem Proving Machine and Slagle's Symbolic Automatic Integrator for elementary calculus problems. (Both Gelernter's and Slagle's programs are landmarks of AI and mechanical theorem-proving and are described in Computers and Thought, [13]).

However, many researchers were more interested in the "ends" than the "means" and launched a no-holds barred attack on the problem of building a program to determine if a propositional formula, and more generally, a first-order formula, was a theorem or, equivalently (thanks to Goedel), valid.

In the early years -- the late 50's and early 60's -- the field was dominated by logicians who pursued quite different approaches to the theorem-proving problem. Among the early researchers were Wang, Gilmore, Davis, Putnam, and Prawitz.

Then, in 1965, J.A. Robinson published the paper "A Machine-Oriented Logic Based on the Resolution Principle" [30]. The resolution principle's simplicity and elegance made it a very attractive mechanism.

Suppose we wished to prove the following theorem of first-order predicate calculus:

$$[\forall X \forall Y\ P(X,Y) \rightarrow Q(Y) \quad \& \quad \forall X \exists Y\ P(X,Y)$$

$$\&$$

$$\forall X\ Q(X) \rightarrow Q(G(X))]$$

$$\rightarrow$$

$$\exists X\ \ Q(G(G(X)))$$

To apply resolution we actually work on the negation of the problem and attempt to derive a contradiction. The negation of the formula above is that the first three hypotheses are true and the conclusion is false. Then we put the conjecture into conjunctive normal form, using Skolem functions to eliminate the existential quantifiers. The result is the following conjunction of disjunctions:

```
-P(X,Y) v Q(Y)

      &

P(Z,F(Z))

      &

-Q(U) v Q(G(U))

      &

-Q(G(G(V)))
```

Finally, Robinson writes this as a set of clauses. A clause is a set of literals, each literal being an atom or negated atom.

```
{-P(X,Y) Q(Y)}

{P(Z,F(Z))}

{-Q(U) Q(G(U))}

{-Q(G(G(V)))}
```

Having distilled the problem down to this simple but universal notation we can now apply the "resolution principle": Consider any two clauses in the set, rename their variables so they have no variable in common, and then consider each literal of one clause against each literal of the other. If the two literals have opposite signs and there exists a substitution that makes their atoms identical, instantiate both clauses with the most general such substitution, delete the two (now complementary) literals from the two instantiated clauses and union the two resulting sets together. The resulting clause is a resolvent of the two parent clauses and should be added to the set of clauses. Repeat this process indefinitely. Should the empty clause ever be formed, the original set of clauses was unsatisfiable -- i.e., the original quantified formula is a theorem.

Perhaps more important than resolution itself was Robinson's "unification algorithm" which is a way to determine either the most general substitution that makes two terms identical or that no such substitution exists. For example, the unification algorithm

determines that P(X,F(X)) and P(A(),Z) are unified by replacing X by A() and Z by F(A()), while P(X,F(X)) and P(X,X) have no common instance.

Here is a resolution proof of the example theorem above:

1.	{ -P(X,Y) Q(Y) }	given
2.	{ P(Z,F(Z)) }	given
3.	{ -Q(U) Q(G(U)) }	given
4.	{ -Q(G(G(V))) }	given
5.	{ Q(F(Z)) }	resolving 1 & 2
6.	{ Q(G(F(Z))) }	resolving 3 & 5
7.	{ Q(G(G(F(Z)))) }	resolving 3 & 6
8.	{ }	resolving 4 & 7

Despite its simplicity, resolution with factoring -- a rule permitting the instantiation of a clause so as to cause two literals in it to become identical -- is a sound and complete inference procedure for first-order predicate calculus. For more details the reader should see [10], [24], or [31].

Note how easily a resolution theorem-prover can be implemented. Clauses may be represented as lists of literals. The basic operation on a Resolution Logic Machine is:

(1) Choose a clause to factor or two clauses to resolve upon.

(2) Form all possible factors or resolvents and add them to the set of clauses.

(3) If any clause is empty, report that the original set was unsatisfiable.

(4) Otherwise, repeat from step(1).

As one might gather from the above description, the only difficult problem is deciding which clauses to choose in any given round. This is called the search strategy and is the hard problem confronting the serious implementor of a resolution theorem-proving.

There are two classic search strategies. One, called breadth first, constructs all the resolvents from anong the initial set S

before adding them to S to form the new set S', and then iterates on the set S'. Thus, the so-called "search tree" -- the tree of all possible resolvents -- is grown in horizontal layers. The other common variation is called depth first, in which one prefers as a parent the most recently produced clause. In a depth first search, long branches of the search tree are grown first.

It is fair to say that very few resolution theorem-provers use either search strategy in the rigid way they are defined above. It is also fair to say that resolution is not the only part of theorem-proving concerned with search strategy. The consideration of search strategy dominates the implementation of a resolution theory-prover largely because resolution has distilled the theorem-proving process down to where there is very little else to do. But every theorem-proving machine (for sufficiently rich logics) stands or falls on its ability to make the right choices at the right time.

To the criticism that resolution was "unnatural" (to many people) the response was similar to Minsky's later defense of the attempt to build an intelligent machine (paraphrase) : If you wanted to build a machine that flies, would you cover it with feathers? If you wanted to build a machine that thinks, would you use meat? During the late 60's the vast majority of published work on mechanical theorem-proving was resolution based.

But saying that the vast majority of the published work was resolution based is not to say that all the resolution researchers were working on the same idea. The very simplicity of resolution encouraged its elaboration. Resolution was restricted, refined, and extended. There was (in no particular order) unit resolution, hyperresolution, linear resolution, and paramodulation. There was linear paramodulation and hyperparamodulation. There was E-resolution, OL-resolution, P1-resolution, SL-resolution, V-resolution and P-hyperparamodulation.

In short, the late 60's were an exciting time in the history of mechanical theorem-proving. There were three (causally related) reasons:

(1) technological improvements brought a tremendous increase in the computer power available.

(2) the economy boomed and made money available for computer science research in previously unheard of quantities -- much of it funnelled through the Advanced Research Projects Agency (ARPA) of the U.S. Defense Department, and

(3) Artificial Intelligence emerged as an endeavor that captured the imaginations of many researchers (and funding agencies) and, theoretically at least, theorem-proving could solve many of the hard problems in AI. For example, several typical AI problems such as natural language understanding, robotics problem solving, and question answering systems could be cast in the framework of first-order predicate calculus problems and solved with sufficient theorem-proving power.

While resolution theorem-proving did not directly receive very much of the money channeled to AI, it benefited greatly from the availability of computer power and the interest in mechanical problem solving generated by AI

Of course, not all researchers pursued resolution, even in its heyday. The interested reader should see, for example, the work of Bledsoe [2] on set theory and Bledsoe, Boyer and Henneman [3] on proofs of limit theorems in real analysis. During this same time, the field of "symbolic manipulation" matured to the point where programs were able to aid physicists and engineers in algebraic simplification and integral calculus. See the review by Moses [28].

In the mid-70's the excitement over resolution declined because researchers began to realize that the paradigm established by Robinson -- formulate a restriction of resolution and prove that it is complete -- produced a plethora of theoretical papers but very few successful mechanical theorem-provers.

Many people attributed this disparity to the "unnaturalness" of resolution and began to pursue new directions. At about the same time, new AI programming languages began to catch on (e.g., PLANNER). For a while in the early 70's controversy raged between those on opposite sides of the question: "Is it better to use 'declarative' or 'procedural' encodings of knowledge?" This controversy has since died out, partially because PLANNER and its descendants did not really solve the hard problems and partially because people like Kowalski and Hayes successfully argued that predicate calculus could be used as a programming language and made to perform as well (or badly) as "conventional" languages like PLANNER.

In my view, the disparity between the number of publications and the number of successful implementations was due to inadequate attention to search strategy. While the search strategy problem was certainly recognized by all, it was more or less left to the "hackers" who put together theorem-provers. It is certainly safe to say that most researchers hoped that victory would be achieved without the invention of messy, ad hoc heuristics. That hope has waned considerably since the early 70's.

During the 70's theorem-proving research was supported mainly by the emerging fields of programming language design and program verification. The main application in programming language design has been the implementation of efficient "interpreters" (i.e., theorem-provers) for nondeterministic predicate calculus programs. The interested reader should see Kowalski's article "Predicate Calculus as a Programming Language" [21], and the work on implementing such a language by Colmerauer and Roussel of the University of Maresille [12], [32], and Warren at the University of Edinburgh [37]. It is interesting to note that in this application search strategy is often less important than in general purpose theorem-proving because the user of the theorem-prover can often constrain the search space by appropriately formulating his "programs".

The theorem-proving research supported by program verification has been both more and less traditional -- more traditional in the sense that the goal is to mechanize mathematics and less traditional in the sense that the approaches used are often radically different from those suggested by resolution. The basic idea -- as will be elaborated in the third lecture -- is that it is easy to transform the question "Is this program correct?" into the question "Are these formulae theorems?" The formulae are then submitted to a mechanical theorem-prover for proof. A theorem-prover for program verification must be good at deriving theorems from a large data base containing facts that may be instantiated and chained together -- just as the AI applications demanded. But, in addition, program verification added some new demands:

(1) The proof of the conjectures produced by program verifiers often require induction. Why? Because those conjectures usually involve inductively constructed mathematical objects (e.g., integers, sequences, trees) and inductively defined concepts (e.g., addition, permutation, fringe).

(2) Program verification has caused the construction of new logical theories in which the semantics of program are expressed.

(3) Program verification aims at putting the theorem-prover in the hands of a "user" who is considered willing to help the theorem-prover but who is not logically infallible. For example, to specify his program the user may need to define previously unstudied mathematical concepts (e.g., majority vote). The addition of axioms purported to describe the properties of such concepts must not be taken lightly. Experience has shown that users are notoriously bad at getting the details right when dealing with concepts outside of their traditional training -- and the accidental production of an inconsistent set of axioms may lead to "proofs" of incorrect programs whose specifications do not

even involve those axioms. On the other hand, experience has shown that many users have excellent intuitions about why things are true and can be of great help in guiding the system to a proof.

Because of these demands theorem-proving research in the 70's has branched out considerably.

Let me merely list some of the main themes of theorem-proving in the 70's:

(1) The construction of proof checkers and interactive theorem-provers. See for example the FOL system of Weyhrauch [38] or Jutting's description of the use of the AUTOMATH system to proof check all the Landau's text on the development of elementary mathematics from Peano axioms to the reals [20].

(2) The construction of theorem-provers for decidable theories, such as Presburger arithmetic and "data structures". See for example the work of Bledsoe [4], Shostak [33] and Oppen [29].

(3) The construction of theorem-provers or proof-checkers for logics other than first-order predicate calculus. For example, our work [7] is based on a quantifier free logic with recursive functions and induction. The Edinburgh LCF system [16] is based on Scott's logic and Litvintchouk and Pratt's system is based on modal logic [23].

(4) The application of rewrite rules to simplify formulas and the study of the theoretical properties of such "rewrite systems". See the survey paper by Huet and Oppen [19].

(5) The study of "metatheoretic extensibility" -- the use of a theorem-prover to prove the correctness of extensions. See below and [8].

(6) The further study of resolution and proof procedures suggested by resolution. See for example the proceedings of the latest Workshop on Automatic Deduction or Kowalski's "connection graph" proof procedure suggested by the failure modes of resolution [22].

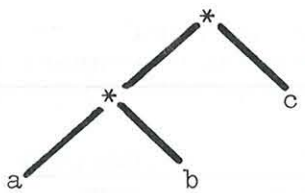
Rather than try to summarize each of these fields I will, in my next lecture, acquaint you with how one state-of-the-art theorem-prover works and what are the current limits of its abilities.

II THE BOYER-MOORE THEOREM-PROVER

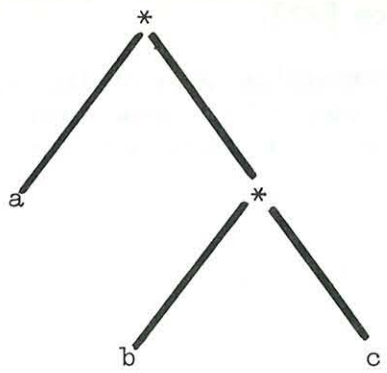
For the past nine years Bob Boyer and I have been developing an automatic theorem-prover capable of constructing inductive proofs. The development of the theorem-prover is being sponsored by NSF Grant MCS-7904081 and ONR Contract N00014-75-C-0816. The theorem-prover deals with a quantifier free first-order logic. In addition to modus ponens, instantiation, and substitution of equals for equals, the logic provides for the axiomatic introduction of new "types" of inductively constructed objects (e.g., integers, sequences, graphs) the definition of new mathematical functions (e.g., prime, permutation, path), and proof by induction on well-founded relations.

The addition of definitional equations purporting to define new functions raises a difficult problem: how can we insure that the new axiom actually defines a function? In our logic we require that for each new definition there exists a "measure" of the arguments of the function and a well-founded relation such that in every "recursive call" in the body, the measure of the argumennts to the call is strictly smaller than the measure of the input arguments. This condition, together with some trivial syntactic requirements, is sufficient to insure that the new axiom is satisfied by one and only one function.

For example, consider the idea of computing the "fringe" of a binary tree. One way to do it is to consider the successive CDR's of the tree and repeatedly transform subtrees of the form:



into the form:



until a is an atom. Using a LISP-like syntax we express this function as:

```
Definition.
(NORMTREE X)
   =
(IF (LISTP X)
    (IF (LISTP (CAR X))
        (NORMTREE (CONS (CAAR X)
                        (CONS (CDAR X) (CDR X))))
        (CONS (CAR X) (NORMTREE (CDR X))))
    (CONS X NIL)).
```

What measure is going down here? Our system is not capable of discovering (on its own) such a measure. However, if the user of our system defines the function:

```
Definition.
(MS X)
   =
(IF (LISTP X)
    (TIMES (SQUARE (MS (CAR X))) (MS (CDR X)))
    1),
```

which is accepted because the size of the argument gets smaller in each call, then the system can prove that (MS X) decreases in both of the recursive calls of NORMTREE in the definition of NORMTREE. Thus, after the introduction of MS and the proof of the two lemmas establishing that it decreases, NORMTREE is accepted by our system as a true definitional equation.

The theorem-prover itself consists of an ad hoc collection of heuristic proof techniques. The two most important ones are simplification and the invention of "appropriate" induction arguments. The system also contains heuristics for eliminating "undesirable" expressions (e.g., X-Y can be eliminated by replacing X with I+Y), the use of equality, generalization, and the elimination of irrelevance.

The simplification routine is driven by conditional rewrite rules derived from axioms, recursive definitions, and previously proved theorems. The system contains fairly sophisticated search strategic heuristics for controlling the expansion of definitions, backwards chaining to establish hypotheses of rewrite rules, permutative rewrites, etc.

The induction routine attempts to find an induction argument that is "appropriate" for the conjecture being proved. Roughly speaking, it attempts to find an n-way case split and some induction hypotheses such that when certain of the recursive functions in the induction conclusion of a given case are expanded, the resulting

recursive calls are involved in the hypotheses for that case. To find -- and justify -- the induction argument, the induction routine analyzes the measures and well founded relations justifying the recursive functions in the conjecture. We have found that the direct analysis of these measures and well-founded relations is simpler than the analysis of the recursive functions themselves and permits the system more often to piece together induction arguments "appropriate" for several functions in the conjecture. The reason for this is that the function definitions frequently contain tests that are irrelevant to the recursions and these tests obscure the correct choice of induction cases.

To illustrate how our system proves theorems, let us consider proving that NORMTREE computes the fringe as defined in the more traditional way by the recursive function FLATTEN:

```
Definition.
(FLATTEN X)
   =
(IF (LISTP X)
    (APPEND (FLATTEN (CAR X))
            (FLATTEN (CDR X)))
    (CONS X NIL)),
```

where APPEND concatenates two lists:

```
Definition.
(APPEND X Y)
   =
(IF (LISTP X)
    (CONS (CAR X)
          (APPEND (CDR X) Y))
    Y).
```

We will prove:

```
(EQUAL (NORMTREE X) (FLATTEN X)).
```

The proof may be briefly sketched as follows: We induct on X, using the measure and well-founded relation justifying NORMTREE, we simplify, using the axioms defining lists and "opening up" certain recursive function "calls", we rerepresent the variables in the problem to simplify the conjecture, use our induction hypothesis (and throw it away to generalize our goal), replace certain terms by variables, restricted to the range of terms replaced, to generalize the problem still more, and then perform a second induction. Below is the complete proof, as discovered and described in English by our system:

```
_PROVE.LEMMA(CORRECTNESS.OF.NORMTREE
                          (REWRITE)
                          (EQUAL (NORMTREE X) (FLATTEN X)))
```

Give the conjecture the name *1.
We will appeal to induction. Two inductions are suggested by terms in the conjecture, both of which are unflawed. So we will choose the one suggested by the largest number of nonprimitive recursive functions. We will induct according to the following scheme:

```
(AND (IMPLIES (NOT (LISTP X)) (p X))
     (IMPLIES (AND (LISTP X)
                   (LISTP (CAR X))
                   (p (CONS (CAAR X)
                            (CONS (CDAR X) (CDR X)))))
              (p X))
     (IMPLIES (AND (LISTP X)
                   (NOT (LISTP (CAR X)))
                   (p (CDR X)))
              (p X))).
```

The inequalities MS.DECREASES.UNDER.CDR and MS.DECREASES.UNDER.ROTATION establish that the measure (MS.X) decreases according to the well-founded function LESSP in each induction step of the scheme. The above induction scheme generates the following three new conjectures:

```
Case 1. (IMPLIES (NOT (LISTP X))
                 (EQUAL (NORMTREE X) (FLATTEN X))),
```

which we simplify, unfolding the definition of NORMTREE and FLATTEN, to:

```
(TRUE).
```

```
Case 2. (IMPLIES
         (AND
           (LISTP X)
           (LISTP (CAR X))
           (EQUAL (NORMTREE (CONS (CAAR X)
                                  (CONS (CDAR X) (CDR X))))
                  (FLATTEN (CONS (CAAR X)
                                 (CONS (CDAR X) (CDR X))))))
         (EQUAL (NORMTREE X) (FLATTEN X))),
```

which simplifies, applying CDR.CONS and CAR.CONS, and opening up FLATTEN and NORMTREE, to:

```
          (IMPLIES
           (AND
            (LISTP X)
            (LISTP (CAR X))
            (EQUAL
```

```
                (NORMTREE (CONS (CAAR X)
                                (CONS (CDAR X) (CDR X))))
                (APPEND (FLATTEN (CAAR X))
                        (FLATTEN (CONS (CDAR X) (CDR X))))))
        (EQUAL (NORMTREE (CONS (CAAR X)
                               (CONS (CDAR X) (CDR X))))
               (APPEND (FLATTEN (CAR X))
                       (FLATTEN (CDR X))))).
```

This simplifies again, applying CDR.CONS and CAR.CONS, and opening up the definition of FLATTEN, to:

```
(IMPLIES
 (AND
    (LISTP X)
    (LISTP (CAR X))
    (EQUAL (NORMTREE (CONS (CAAR X)
                           (CONS (CDAR X) (CDR X))))
           (APPEND (FLATTEN (CAAR X))
                   (APPEND(FLATTEN (CDAR X))
                          (FLATTEN (CDR X))))))
 (EQUAL (NORMTREE (CONS (CAAR X)
                        (CONS (CDAR X) (CDR X))))
        (APPEND (APPEND (FLATTEN (CAAR X))
                        (FLATTEN (CDAR X)))
                (FLATTEN (CDR X))))).
```

Applying the lemma CAR/CDR.ELIM, we now replace X by (CONS Z V) to eliminate (CAR X) and (CDR X) and Z by (CONS W D) to eliminate (CAR Z) and (CDR Z). This generates:

```
(IMPLIES
  (EQUAL (NORMTREE (CONS W (CONS D V)))
         (APPEND (FLATTEN W)
                 (APPEND (FLATTEN D) (FLATTEN V))))
  (EQUAL (NORMTREE (CONS W (CONS D V)))
         (APPEND (APPEND (FLATTEN W) (FLATTEN D))
                 (FLATTEN V)))).
```

We now use the above equality hypothesis by substituting:

```
(APPEND (FLATTEN W)
        (APPEND (FLATTEN D) (FLATTEN V)))
```

for (NORMTREE (CONS W (CONS D V))) and throwing away the equality.

The result is:

```
(EQUAL (APPEND (FLATTEN W)
               (APPEND (FLATTEN D) (FLATTEN V)))
       (APPEND (APPEND (FLATTEN W) (FLATTEN D))
               (FLATTEN V))),
```

which we generalize by replacing (FLATTEN V) by Y, (FLATTEN D) by A, and (FLATTEN W) by U. We restrict the new variables by appealing to the type restriction lemma noted when FLATTEN was introduced. This produces:

```
(IMPLIES (AND (LISTP Y) (LISTP A) (LISTP U))
         (EQUAL (APPEND U (APPEND A Y))
                (APPEND (APPEND U A) Y))),
```

which we will name *1.1.

Case 3.
```
(IMPLIES (AND (LISTP X)
              (NOT (LISTP (CAR X)))
              (EQUAL (NORMTREE (CDR X))
                     (FLATTEN (CDR X))))
              (EQUAL (NORMTREE X) (FLATTEN X))),
```

which we simplify, expanding the definitions of NORMTREE and FLATTEN, to:

```
(IMPLIES (AND (LISTP X)
              (NOT (LISTP (CAR X)))
              (EQUAL (NORMTREE (CDR X))
                     (FLATTEN (CDR X))))
         (EQUAL (CONS (CAR X) (NORMTREE (CDR X)))
                (APPEND (FLATTEN (CAR X))
                        (FLATTEN (CDR X))))).
```

This simplifies again, applying CDR.CONS, CAR.CONS, and CONS.EQUAL, and opening up the functions FLATTEN and APPEND, to:

```
(TRUE).
```

So let us turn our attention to:

```
(IMPLIES (AND (LISTP Y) (LISTP A) (LISTP U))
         (EQUAL (APPEND U (APPEND A Y))
                (APPEND (APPEND U A) Y))).
```

which we named *1.1 above. We will appeal to induction. Three inductions are suggested by terms in the conjecture.

They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

```
(AND (IMPLIES (NOT (LISTP U)) (p U A Y))
     (IMPLIES (AND (LISTP U) (p (CDR U) A Y))
              (p U A Y))).
```

The inequality CDR.LESSP establishes that the measure (COUNT U) decreases according to the well-founded function LESSP in the induction step of the scheme. The above induction scheme produces two new goals:

```
Case 1. (IMPLIES (AND (NOT (LISTP (CDR U)))
                      (LISTP Y)
                      (LISTP A)
                      (LISTP U))
                 (EQUAL (APPEND U (APPEND A Y))
                        (APPEND (APPEND U A) Y))).
```

This simplifies, applying CDR.CONS, CAR.CONS, and CONS.EQUAL, and expanding the definition of APPEND, to:

```
(IMPLIES (AND (NOT (LISTP (CDR U)))
              (LISTP Y)
              (LISTP A)
              (LISTP U))
         (EQUAL (APPEND (CDR U) (APPEND A Y))
                (APPEND (APPEND (CDR U) A) Y))).
```

which again simplifies, opening up the definition of APPEND, to:

```
(TRUE)
```

```
Case 2. (IMPLIES (AND (EQUAL (APPEND (CDR U) (APPEND A Y))
                             (APPEND (APPEND (CDR U) A) Y))
                      (LISTP Y)
                      (LISTP A)
                      (LISTP U))
                 (EQUAL (APPEND U (APPEND A Y))
                        (APPEND (APPEND U A) Y))),
```

which simplifies, applying CDR.CONS, CAR.CONS, and CONS.EQUAL, and opening up the function APPEND, to:

```
(TRUE).
```

That finishes the proof of *1.1, which, consequently, finishes the proof of *1. Q.E.D.

Load average during proof: 1.865178
Elapsed time: 14.509 seconds
CPU time (devoted to theorem proving): 7.727 seconds
IO time: 3.385 seconds
CONSes consumed: 11520

In the proof above the system "discovers" the lemma that APPEND is associative and proves it by the second induction.

The theorem-prover is automatic in the sense that once it begins a proof the user contributes nothing. However, it is interactive in the sense that the user can improve the theorem-prover's behaviour by "teaching" it important relationships and rewrite rules. This "teaching" (which might be more appropriately called "memorization by rote") is accomplished by instructing the theorem-prover to prove lemmas that "inform" it of new conditional rewrite rules, useful measures for the justification of recursions and inductions, etc. For example, had the user previously instructed the system to prove the associativity of APPEND the system would have used that fact in the proof above, leading to a substantially simpler proof.

The user of our system does not have to be trusted. That is, as long as he confines himself to the "rules of the game" (i.e., defining new types and functions and proving new lemmas), the theorem-prover is entirely responsible for the validity of any conjecture it claims is a theorem.

While the user who abides by the rules need not be trusted, an intelligent and well-traained user is indispensable in the proof of difficult theorms because the theorem-prover requires so much carefully prepared groundwork in the form of previously proved lemmas. Much of our research is aimed at reducing some of this burden on the user. However, even at the current rudimentary stage of the system's development, we have found that we (as human users) are quite good at the task required of us (i.e., the strategic planning of proofs encoded in the statement of key lemmas) and are relatively weak at the tasks already performed by the system (the consideration of countless nitty-gritty details).

The system has been used to prove the correctness of a wide variety of programs including:

(1) a "toy" expression compiler [7],

(2) a recursive descent parser (the theorem-prover established the required relationship between "printing" and "reading") [15],

(3) the totality, soundness, and completeness of a decision procedure for propositional calculus [7],

(4) the soundness of an arithmetic simplifier now in routine use in the system [8],

(5) the termination of the TAK function over the positive and negative integers (using a lexicographic measure corresponding to "less than" in omega^3) [26], and

(6) several working FORTRAN programs including the correctness of the fastest known string searching algorithm [9].

I will discuss program verification further in the third lecture.

The most difficult theorem proved to date is the existence and uniqueness of prime factorizations, which was derived entirely from Peano's axioms [7]. While this theorem is not often involved in the correctness proofs of real programs (encryption algorithms excepted), the system's ability to prove it from the ground up is indicative of the theorem-prover's power.

All of the theorems cited above were proved by the same version of the theorem-prover from the same initial set of axioms. The axioms are those defining TRUE, FALSE, IF, and EQUAL, plus the Peano-like axiomatization of the "data types" involved.

Given that the system has some "learning" (or "rote memorization") ability, the question arises: "Is it possible to teach the system new proof techniques that were not anticipated by the designers of the theorem-prover?" Of course, we wish to preserve the soundness of the system, i.e., it should not be possible for the user to render the system unsound by teaching it faulty "proof" techniques.

Since our system is oriented towards proving properties of programs, an obvious approach is for the user to write a new theorem-proving routine to be added to the system, and then have the trusted version of the system prove the new extension correct before encorporating it. Can a system which is inherently inadequate (after all, it is in need of extension) be expected to prove the correctness of a useful extension? We have investigated this problem and believe the answer, for our system, is "yes".

One experiment we performed involved the addition of a simple cancellation routine. Suppose I, J, K and L are nonnegative integers. It is easy to prove that I+J=I+K iff J=K. This is the traditional statement of the cancellation law for addition. But note that this rule cannot be applied to L+J=K+(I+L), because the common term, L, does not occur as the first addend. While we could prove many different versions of the cancellation law, no finite number of rewrite rules can capture the underlying idea: you can cancel any term occurring as an addend on both sides of an equality. How can we teach our system this idea?

We can proceed as follows. Define the function CANCEL on list expressions that, when given an expression representing an equation between two PLUS-trees, returns a new expression with all the common addends deleted, and when given any other expression returns the input expression. To cancel common addends CANCEL computes the fringe of the two PLUS-trees, intersects them, subtracts the intersection from each fringe, and then reconstitutes the remaining lists of terms as right-associated PLUS-trees and equates them. One must be careful to keep in mind that the fringes are bags, not sets, and that duplications have significance (e.g., if A occurs twice on one side and only once on the other, only one A can be cancelled).

Once CANCEL has been defined it can be used as a new proof technique provided we can prove the following "metatheorem": Suppose X is a list structure representing a term in our logic and MEANING is the function that assigns values to such list structures, given an assignment of values to atomic symbols. Then we wish to prove that under all assignments the MEANING of X is equal to the MEANING of (CANCEL X) and (CANCEL X) represents a term. That is, we wish to prove:

```
(IMPLIES (FORMP X)
         (AND (EQUAL (MEANING X A)
                     (MEANING (CANCEL X) A))
              (FORMP (CANCEL X)))).
```

This theorem can be proved by the current system, after the user has had the system prove the rudiments of "bag theory" (e.g., that the difference between two bags is a subbag of the first) and the fundamental relationships induced by MEANING between bag operations and arithmetic (e.g., if Y is a subbag of X then the meaning of the PLUS-tree formed from the bag difference of X and Y is the arithmetic difference of the MEANING of the PLUS-trees formed from X and Y individually).

After proving the correctness of CANCEL, the system can use CANCEL to perform arbitrarily deep cancellations, an ability it did not have before or during the correctness proofs.

Except for the work on "metatheorems" all of the work described here is described in complete detail in our book, [7]. The book describes our formal theory (assuming only that the reader is familiar with propositional calculus and equality) and all of the proof techniques used by our program. The techniques are illustrated with many substantial examples worked by the program. The techniques are described in sufficient detail to permit a student to use them to discover proofs as well as to program a computer to reproduce our results. The work on metatheorems mentioned here is described in complete detail in [8].

Discussion

Professor Paul: Is the strategy used in the proof influenced by the definition of the function?

Dr. Moore: Yes. The definition is used to find a good induction.

Professor Paul: This may not lead to the most efficient proof.

Dr. Moore: Yes but this doesn't matter because we are not interested in the efficiency of the solution.

Professor Paul: I don't believe that the theorem prover would find a new and exciting solution. Interactions with the user may give it enough insight to help it through.

Dr. Moore: In finding a solution the theorem proven may well produce alternative forms of the original definition, and thereby help the user.

Professor Katzenelson: Can you give any figures for execution times and program size?

Dr. Moore: On a KL-10 NORMTREE takes 7.7 seconds (the KL-10 has a speed of 1.6 MIPS) while the hardest case may take 7 minutes. The compiled code takes 70K words, and we have found no proof that requires more than 36K words of free space. Any new facts stored take no space due to a virtual memory system that is used. The information is put on disc.

Professor Dijkstra: Can you describe how the system has evolved over ten years?

Dr. Moore: The system at Edinburgh proved everything from scratch, it did not use previously proved theorems. All inductions were structured, and heuristic rewrite rules were used.

Professor Dijkstra: How do you maintain the consistency of the system?

Dr. Moore: Most of the system's time is spent proving theorems. We look for heuristics which would make the proof of a theorem easier. If the system is modified it will be asked to prove all the theorems used publicly. This is good because a modification may just cause the search space to shift, which may in turn lead previously proved theorems to fail.

III PROGRAM VERIFICATION

One important application of mechanical theorem-proving is formal program verification. Formal program verification is a relatively new approach to the program reliability problem in which programs are formally specified and proved to meet those specifications.

There are many different approaches to program verification, but they have one thing in common: they reduce the questions "Is this program correct?" to the question "Are these formulae theorems?" Since the objective is the elimination of errors, the process of generating and proving the formulae must be mechanized.

In this talk I will illustrate several program verification methods and use them to derive conjectures proved by the theorem-prover described above. Three methods will be explained: the functional method, the interpreter method, and the inductive assertion method. Each will be applied to the same "toy" problem. Then I will briefly discuss and illustrate how we are using our theorem-prover to prove the correctness of ANSI FORTRAN programs, where we handle such difficult problems as aliasing, global COMMON, and arithmetic overflow.

A. A Toy Example

Let us consider a simple assembly language program to sum the numbers from 1 to I:

```
0       MOVE AC, 0              ;set AC to 0
1       SKIPNE I                ;skip next if I not 0
2       STOP                    ;stop
3       ADD AC, I               ;set AC to AC+I
4       SUBI I, 1               ;set I to I-1
5       JUMP 1                  ;jump to instruction 1
```

We wish to prove that when this program is executed the final value of AC is (i*i+1)/2 where i is the initial value of I. We assume i is a nonnegative integer.

We will consider three different methods of attaching semantics to this program. It is advantageous in all three cases to first introduce the recursive function that sums the integers from M to N:

```
(SIGMA M N)
   =
(IF (LESSP M N)
    (PLUS N (SIGMA M (SUB1 N)))
    0).
```

For example, (SIGMA 3 7) is 7+6+5+4. It is also worthwhile proving the general result that (SIGMA 0 I) is (I*(I+1))/2. This is proved by the theorem-prover described above, using induction on I. Having proved this lemma, it is now sufficient to establish that our 6-line assembly program computes (SIGMA 0 I).

B. The Functional Method

The first method we will consider, often called the "functional" or "McCarthy" method [25], is to view one's program as a mathematical function from input states to output states and to prove the correctness of the resulting function.

Formally speaking, states are n-tuples specifying the values of the global and local variables of the program. In general, each loop in the program is transformed into a recursively defined function on states.

Consider the assembly language program above. The state is given by <I,AC>. The program has one loop, starting at the SKIPNE instruction at location 1. Provided I is not zero, the program sets AC to AC+I, decrements I, and repeats. When I is zero, the program halts. Since we are interested only in the final value of AC and not that of I we transform this loop into a function from I and AC to the final value of AC:

```
Definition.
(LOOP I A)
    =
(IF (ZEROP I)
    AC
    (LOOP (DIFFERENCE I 1)
          (PLUS AC I))).
```

Since the program enters the loop after setting AC to 0, the entire program is functionally equivalent to (LOOP I 0).

As McCarthy noted, such a transformation can be carried out mechanically. The transformation mechanism is an encoding of a semantics of the source language. For more details see [25], [27], and [5]. The admission of LOOP as a function in our logic establishes the termination of the program. The conjecture we wish to prove is that (LOOP I 0) is equal (SIGMA 0 I). As often is the case, it is easier to prove the following more general fact about LOOP:

```
(EQUAL (LOOP I AC)
       (PLUS AC (SIGMA 0 I))),
```

for all numeric AC. The proof of this generalization is straightforward by induction on I and can be constructed by the theorem-prover described above.

C. The Interpreter Method

A second method for formalizing the properties of a program is to specify formally an interpreter for the programming language. This is akin to "denotational semantics" [17].

In this case we must specify the "hardware" that runs our 6-line program. We will do so by writing a recursive function, EXEC, that takes three arguments: the program counter, pc; a memory, mem, mapping integer addresses to their values; and a clock, clk, that ticks once every time we execute a jump instruction. The clock is used to make EXEC a total recursive function. EXEC is an accurate if somewhat simple formalization of the idea of a stored program computer. Each instruction is a list containing an "opcode" and some "arguments" and will occupy one location in memory. In our example, the program will be loaded into memory locations 0 through 5, we will use locations 6 and 7 for the variables I and AC.

EXEC operates as follows. If the clock is 0, EXEC returns an error signal. Otherwise, EXEC fetches the contents of location pc in mem and decodes it as an instruction, obtaining the opcode, op, and two operands arg1 and arg2. If op is STOP, EXEC returns the final memory configuration. Otherwise, EXEC determines new values for pc, mem, and clk based on op and the operands and recurses on those new values. For example, if op is JUMP, EXEC recurses, replacing pc by arg1 and decrementing clk. If op is ADD, EXEC recurses replacing pc by pc+1 and mem by

```
(SET arg1
     (PLUS (GET arg1 mem) (GET arg2 mem))
     mem).
```

Thus, after executing (ADD arg1 arg2) the "new" memory is that obtained by adding the contents of address arg1 to that of address arg2 in the old memory and then setting the contents of address arg1 to that sum. (GET and SET are defined functions that operate on finite sequences denoting the contents of successive memory locations.) The other opcodes used in our program are handled similarly.

Once EXEC is defined we can state the correctness of our program as the following conjecture:

```
(IMPLIES (AND (EQUAL MEM
                     (APPEND '((MOVEI 7 0)
                               (SKIPNE 6)
                               (STOP)
                               (ADD 7 6)
                               (SUBI 6 1)
                               (JUMP 1))
                             REST))
              (EQUAL I (GET 6 MEM))
              (NOT (LESSP CLK I)))
         (EQUAL (GET 7 (EXEC 0 MEM CLK))
                (IF (ZEROP CLK)
                    (GET 7 MEM)
                    (SIGMA 0 I))))
```

This formula says: If locations 0-5 of MEM contain the program in question and if I is the contents of location 6 in MEM and is less than or equal to CLK, then the value of location 7 in the memory obtained by executing the program starting at pc 0 in MEM with CLK is (SIGMA 0 I). (If CLK is zero, then the value of location 7 after execution is the original value of location 7.)

This conjecture can be proved by our theorem-prover. The proof requires that the system first prove a lemma: provided there is sufficient time on the clock, EXEC computes the sum of AC and (SIGMA 0 I) if started at location 1 (instead of location 0).

D. <u>The Inductive Assertion Approach</u>

We now move on to an illustration of the "inductive assertion" or "Floyd/Hoare" method [14], [18]. The basic idea is to attach to the input, output, and every loop of the program an assertion that describes the state of the machine each time execution reaches the annotated point. One may then analyze the finite number of execution paths between any two assertions and generate a set of formulas called "verification conditions" that establish that each assertion holds each time it is encountered. The verification condition generator ("vcg") is an encoding of a sematics of the programming language.

The annotation of our example program above is as follows. Suppose K is the initial value of I. The "input assertion" is T; that is, we put no constraints on I initially. The "output assertion," at the STOP instruction at location 2, is that AC is equal to (SIGMA 0 K). The "loop invariant", at the SKIPNE instruction at location 1, is that AC is equal to (SIGMA I K) and $I \leq K$. By exploring the paths through the program (using some formal specification of the effects of each instruction) we generate three verification conditions to prove:

(1) The loop assertion is true when first encountered:

```
(AND (EQUAL 0 (SIGMA K K))
     (LESSEQP K K)).
```

This is just the loop assertion with I replaced by its initial value, K, and AC replaced by 0.

(2) If the loop assertion holds and we go around the loop, then the loop assertion holds for the new values of I and AC:

```
(IMPLIES (AND (EQUAL AC (SIGMA I K))
              (LESSEQP I K)
              (NOT (ZEROP I)))
         (AND (EQUAL (PLUS AC I)
                     (SIGMA (DIFFERENCE I 1) K))
              (LESSEQP (DIFFERENCE I 1) K))).
```

(3) If the loop assertion holds and we exit the loop, then the output assertion holds.

```
(IMPLIES (AND (EQUAL AC (SIGMA I K))
              (LESSEQP I K)
              (ZEROP I))
         (EQUAL AC (SIGMA 0 K))).
```

These three formulae establish that when the program terminates AC is (SIGMA 0 K). They do not establish termination, although that can be done by a similar path analysis. These verification conditions can be proved by our theorem-prover.

E. Comparisons

The three program verification methods sketched are more striking in their similarities than in their differences.

First, it should be noted that the introduction of SIGMA simplifies the conjectures produced by all three methods. A more commonly used specification style - at least when the inductive assertion method is chosen - is to restrict oneself to "primitives" such as addition, multiplication, and division built into the system. In this example this makes the verification conditions more difficult to prove because one is simultaneously grappling with the fundamental mathematical fact that (SIGMA 0 I) is (I*(I+1))/2 and with a particular algorithm for computing (SIGMA 0 I).

Second, all three methods require some creative step beyond the mere specification of the input/output relation. In the functional method, this creative step is the generalization of:

(EQUAL (LOOP I 0) (SIGMA 0 I)))

to

(EQUAL (LOOP I AC) (PLUS AC (SIGMA 0 I))).

In the interpreter method, the creative step is the statement of the lemma that when EXEC starts executing at location 1 and runs to normal completion, the answer is (PLUS AC (SIGMA 0 I). In the inductive assertion method, the creative step is the invention of the loop invariant that AC is (SIGMA I K).

It should be noted that with the functional and interpreter methods the creativity occurred after the problem had been cast mathematically and while a proof was being sought. That is, the creative steps were just generalizations in the mathematical sense: given to prove p we decided to prove q, where q implies p. In the inductive assertion method, we were obliged to think about q before the problem could be stated without reference to the program text. Thus, when the former methods are applied, this creative aspect of the problem is just a theorem-proving problem; when the inductive assertion method is applied, this creative step is generally regarded as a specification problem.

For our 6-line assembly language program, the theorems generated by the functional method are the easiest to prove, with the inductive assertion method second and the interpreter method a distant third. Of course, nothing in general should be inferred from this ranking.

For example, applying the functional method to messier programs - especially programs manipulating large global data structures - often produces unmanageably large recursive equations; in such cases the inductive assertion method can often be used to segment the program and isolate side-effects.

On the other hand, the interpreter method has an elegance the other two lack because our program was proved correct with respect to a formal programming language sematics defined entirely within the logic itself rather than in some extralogical axioms or ad hoc program transformations. Furthermore, the interpreter method as it was applied here dealt with a problem neither of the other two methods could possibly handle: the instructions were being fetched from a memory that was being modified by the execution of the program. While the program does not happen to modify itself, consideration of that possibility vastly complicates the proof. When the hardware method is formalized so that the program is in "read only" memory (i.e., a memory held constant in the EXEC recursion) the interpreter-based proofs are no more complicated than the inductive assertion style proofs.

F. Toys v. Reality

The preceding sketches were meant to summarize several different approaches to program verification and to illustrate the role of theorem-proving in each of them. However, all three sketches dealt with a toy problem. We did not describe a useful programming language. We ignored many difficult problems of programming language design (e.g., data structures, subroutine calls, aliasing). We ignored many difficult problems of programming language implementation (e.g., arithmetic overflow, array bounds violations, undefined variables). In short, the toy problem discussed here bear about as much resemblance to real programming problems as $E=Mc^2$ does to a nuclear power plant. Rather than attempt to describe how these problems can be dealt with I will simply "advertise" and illustrate how we have dealt with them in the context of one real programming language.

We have implemented a verification condition generator for a subset both of FORTRAN 66 [35] and FORTRAN 77 [1]. While constraints are placed on the language that are not found in the ANSI specifications, our language is a true subset in the sense that a processor that correctly implements either FORTRAN correctly implements our language. The development of the FORTRAN verification condition generator was supported by ONR Contract N00014-75-C-0816.

Unusual features of our system -- aside from our choice of FORTRAN and our use of a quantifier free specification language -- include a syntax checker that enforces all our syntactic restrictions on the language, the thorough analysis of aliasing, the generation of verification conditions to prove termination, and the generation of verification conditions to ensure against such run-time errors as array-bound violations and arithmetic overflow.

Although our syntax checker and verification condition generator handle programs involving finite precision real arithmetic, we have not yet formalized the semantics of those operations and hence cannot mechanically verify programs that operate on REALs.

We define our subset precisely in [9] and specify the verification conditions we generate. The following description of our work is extremely informal.

The input to our verification condition generator must include not only the subprogram (function or subroutine) to be verified, but also all subprograms referenced somehow by the candidate subprogram. Each referenced subprogram must have been previously specified and verified.

The FORTRAN statements in our subset are:

Arithmetic assignment	DO
Logical assignment	DIMENSION
GO TO assignment	COMMON
Unconditional GO TO	INTEGER
Assigned GO TO	REAL
Computed GO TO	DOUBLE PRECISION
Arithmetic IF	COMPLEX
CALL	LOGICAL
RETURN	EXTERNAL
CONTINUE	Statement function
STOP	FUNCTION
PAUSE	SUBROUTINE
Logical IF	END

Our subset does not include the following FORTRAN 77 statements:

BACKSPACE	FORMAT
BLOCK DATA	IMPLICIT
Block IF	INQUIRE
CHARACTER	INTRINSIC
Character assignement	OPEN
CLOSE	PARAMETER
DATA	PRINT
ELSE	PROGRAM
ELSEIF	READ
ENDFILE	REWIND
ENDIF	SAVE
ENTRY	WRITE
EQUIVALENCE	

For those statements in our subset we enforce all of the restrictions of both FORTRAN 66 and 77; furthermore, we enforce some additional restrictions. Some of our restrictions are:

Every expression using infix operators must be fully parenthesized. For example, either (A + (B + C)) or ((A + B) + C) must be written instead of A + B + C. The precise order of combination affects the analysis of overflow.

Subroutines and functions may not be passed as arguments to subprograms.

In a CALL statement or function reference, if the actual argument is an array, then the corresponding argument must be an array of the same number of dimensions.

Function subprograms may not side-effect their arguments or anything in COMMON.

No call of a subroutine may pass an entity to a subroutine that might violate the strict aliasing restrictions of FORTRAN. For example, if a subroutine has two arguments and possibly smashes the first, then that subroutine may not be called with the same array passed in both arguments nor may an array in COMMON be passed as the first argument if the subroutine "knows" about the COMMON block, even via subprograms.

While some of our restrictions may appear radical to those unfamiliar with the details of the FORTRAN specifications, many of the most severe (e.g., prohibition of side-effects in FUNCTIONs and aliasing in SUBROUTINEs) are in fact closely related to restrictions in both the 1966 and 1977 specifications. Many of the restrictions in the ANSI specifications were motivated by the desire to encourage the implementation of correct optimizing compilers and -- while the restrictions are not as elegantly stated as they might have been -- it could be argued that FORTRAN 66 was several years ahead of its time. In [9] we compare our restrictions to those of the ANSI specifications. All of our restrictions are enforced by our system in the sense that programs violating these restrictions are rejected by the verification condition generator.

We make the following claim about our system. If a FORTRAN subprogram is accepted by our syntax checker, the verification conditions are proved, and the program can be loaded onto a FORTRAN processor that meets the ANSI specification of FORTRAN and satisfies certain parameterized constraints on the accuracy of arithmetic, then any invocation of the program in any environment satisfying the input condition of the program will terminate without run-time errors and produce an environment satisfying the output condition of the program.

We have used the theorem-prover to prove the verification conditions produced for several working FORTRAN programs, including a FORTRAN implementation of the Boyer-Moore fast string searching algorithm, and several subprograms performing "big number" arithmetic operations on arrays of integers regarded as numbers in a large base (e.g., 2^{18}).

G. A FORTRAN Example

In a 1977 Communications of the ACM article [6], we describe an algorithm for finding the first occurrence of one character string, PAT in another, STR. The algorithm is currently the fastest known way to solve this problem on the average. Our algorithm has two unusual properties. First, in verifying that PAT does not occur within the first i characters of STR the algorithm will typically fetch and look at fewer than i characters. Second, as PAT gets longer the algorithm speeds up. That is, the algorithm typically spends less time to find long patterns than short ones. In this section we briefly describe the verification of a FORTRAN version of the algorithm. A more complete description may be found in [9].

The idea behind the algorithm is illustrated by the following example. Suppose we are trying to find PAT in STR and, having scanned some initial part of STR and failed to find PAT, are now ready to ask whether PAT occurs at the position marked by the arrow below:

```
PAT:                    EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE
                        ↑
```

Instead of focusing on the left-hand end of the pattern (i.e., on the "E" indicated by the arrow) the algorithm considers the right-hand end of the pattern. In particular, the algorithm fetches the "I" in the word "SIMPLE". Since "I" does not occur in PAT, the algorithm can slide the pattern down by seven (the length of PAT) without missing a possible match. Afterwards, it focuses on the end of the pattern again, as marked by the arrow below.

```
PAT:                            EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE
                                ↑
```

In general, as the next step would suggest, the algorithm slides PAT down by the number of characters that separate the end of the pattern from the last occurrence in PAT of the character, c, just fetched from STR (or the length of PAT if c does not occur in PAT). In the configuraion above, PAT would be moved forward by five characters, so as to align the "X" in PAT with the just fetched "X" in STR.

If the algorithm finds that the character just fetched from STR matches the corresponding character of PAT, it moves the arrow backwards and repeats the process until it either finds a mismatch and can slide PAT forward, or matches all the characters of PAT.

The algorithm must be able to determine efficiently for any character c, the distance from the last occurrence of c in PAT to the right-hand end of PAT. But since there are only a finite number of characters in the alphabet we can preprocess PAT and set up a table that answers this question in a single array access.

The reader is referred to [6]for a thorough description of an improved version of the algorithm that can be implemented so as to search for PAT through i characters of STR and execute less than i machine instructions, on the average. In addition, [6]contains a statistical analysis of the average case behaviour of the algorithm and discusses several implementation questions.

A FORTRAN version of the algorithm is exhibited below. The subroutine FSRCH is the search algorithm itself: it takes five arguments, PAT, STR, PATLEN, STRLEN, and X. PAT and STR are one-dimensional adjustable arrays of length PATLEN and STRLEN

respectively. X is the dummy argument into which the answer is smashed. The answer is either the index into STR at which the winning match is found, or else it is STRLEN+1 indicating no match exists.

FSRCH starts by CALLing the subroutine SETUP, which preprocesses PAT and smashes the COMMON array DELTA1. DELTA1 has one entry for each character code in the alphabet. SETUP executes in time linear in PATLEN. It initializes DELTA1 as though no character occurred in PAT and then sweeps PAT once, from left to right, filling in the correct value of DELTA1 for each character occurrence, as though that occurrence were the last occurrence of the character in PAT. Thus, if the same character occurs several times in PAT (as "E" does in "EXAMPLE") then its DELTA1 entry is smashed several times and the last value is the correct one.

```
      SUBROUTINE FSRCH(PAT, STR, PATLEN, STRLEN, X)
      INTEGER DELTA1
      INTEGER PATLEN
      INTEGER STRLEN
      INTEGER PAT
      INTEGER STR
      INTEGER I
      INTEGER J
      INTEGER C
      INTEGER NEXTI
      INTEGER X
      INTEGER MAXO
      DIMENSION DELTA1(128)
      DIMENSION PAT(PATLEN)
      DIMENSION STR(STRLEN)
      COMMON /BLK/DELTA1
      CALL SETUP(PAT, PATLEN)
      I = PATLEN
200   CONTINUE
      IF ((I.GT.STRLEN)) GO TO 500
      J = PATLEN
      NEXTI = (1+I)
300   CONTINUE
      C = STR(I)
      IF ((C.NE.PAT(J))) GO TO 400
      IF ((J.EQ.1)) GO TO 600
      J = (J-1)
      I = (I-1)
      GO TO 300
400   I = MAXO((I+DELTA1(C)), NEXTI)
      GO TO 200
500   X = (STRLEN+1)
      RETURN
600   X = I
      RETURN
      END
```

```
      SUBROUTINE SETUP(A, MAX)
      INTEGER DELTA1
      INTEGER A
      INTEGER MAX
      INTEGER I
      INTEGER C
      DIMENSION DELTA(128)
      DIMENSION A(MAX)
      COMMON /BLK/DELTA1
      DO 50 I=1, 128
      DELTA1(I) = MAX
50    CONTINUE
      DO 100 I=1, MAX
      C = A(I)
      DELTA1(C) = (MAX-I)
100   CONTINUE
      RETURN
      END
```

To specify the input and output assertions FSRCH we must introduce the mathematical concepts of (a) a sequence being a "character string" on a given sized alphabet, (b) the initial segments of two strings "matching", and (c) the leftmost match of PAT in STR. Below we give the definitions of these mathematical functions.

```
Definition.
(STRINGP A I SIZE)
   =
(IF (ZEROP I)
    T
    (AND (NUMBER (ELT1 A I))
         (NOT (EQUAL (ELT1 A I) 0))
         (NOT (LESSP SIZE (ELT1 A I)))
         (STRINGP A (SUB1 I) SIZE)))

Definition.
(MATCH PAT J PATLEN STR I STRLEN)
   =
(IF (LESSP PATLEN J)
    T
    (IF (LESSP STRLEN I)
        F
        (AND (EQUAL (ELT1 PAT J) (ELT1 STR I))
             (MATCH PAT
                    (ADD1 J)
                    PATLEN STR
                    (ADD1 I)
                    STRLEN))))
```

```
Definition.
(SEARCH PAT STR PATLEN STRLEN I)
   =
(IF (LESSP STRLEN I)
    (ADD1 STRLEN)
    (IF (MATCH PAT 1 PATLEN STR I STRLEN)
        I
        (SEARCH PAT STR PATLEN STRLEN
                (ADD1 I)))))
```

For example, (MATCH PAT J PATLEN STR I STRLEN) determines whether the characters of PAT in position J through PATLEN are equal to the corresponding characters of STR starting at position I and not exceeding STRLEN. MATCH is recursive. That is, provided $J \leq$ PATLEN and $I \leq$ STRLEN, MATCH checks that the J^{th} character of PAT is equal to the I^{th} character of STR and, if so, requires that there be a MATCH starting at positions I+1 and J+1. The recursive function SEARCH is the mathematical expression of the naive string searching algorithm. (SEARCH PAT STR PATLEN STRLEN I) is the least i, $I \leq i \leq$ STRLEN, such that a MATCH with PAT occurs at position i, or STRLEN+1 if no MATCH occurs.

The input specification for FSRCH includes the assertion that PAT and STR are both strings on the alphabet from 1 to 128. The output assertion for FSRCH is that whenever it exists, X is set to (SEARCH PAT STR PATLEN STRLEN 1). The loop invariants for FSRCH are expressions in terms of MATCH and SEARCH, asserting that (at label 200) the winning occurrence of PAT in STR has not yet been found and (at label 300) that a partial match has been established between the terminal substring of PAT and part of STR. The verification condition generator produces some 50 theorems that must be proved to establish that these assertions hold, that both SETUP and FSRCH terminate, and that no run-time errors occur. For example, the statement, at location 400 in FSRCH:

```
I = MAXO((I+DELTA1(C)),NEXTI)
```

requires that we prove (1) C is defined, (2) C is a legal index into DELTA1, (3) DELTA1(C) is defined, (4) I is defined, (5) I+DELTA1(C) does not cause an overflow, and (6) NEXTI is defined. The proof that I+DELTA1(C) does not cause an overflow requires that we put an additional input assertion on FSRCH, namely that the sum of lengths of PAT and STR be expressible on our machine.

Discussion

Dr. Henderson: If you are working on a contract for someone else, how much control do you have over conjectures?

Dr. Moore: Historically great control, but clients increasingly want total control. Often I have only to give the information that allows a particular theorem to be proved, given the program and the specification.

Professor Rogers: What is the largest program you have proved?

Dr. Moore: This depends on who is using it. It also depends on which point you start from, i.e. from axioms or from scratch. The largest program is five pages of dense FORTRAN. It has also been used to prove the security of the kernal of an operating system. That code is about one thousand lines long.

Mr. Grossman: Is it possible to handle asynchronous programs with interrupts?

Dr. Moore: There are people looking at this. I don't really know what the answer is, I'm more interested in mechanical theorem proving. It is frontier work to formalise it at all; modal and temporal logic come in all the time.

Dr. Larcombe: Can a theorem prover prove itself correct?

Dr. Moore: No, in a certain sense. If a man says he always tells the truth, what do you think? What do you think if he says he never tells the truth?

The following proof is conceivable however. Implement a simple theorem prover and then extend it. The extended system may be proved using the simple one, and the process is repeated until the desired theorem prover is produced. The simple system is about a page long and so could be proved by getting say ten mathematicians to agree it was correct.

Professor Randell: Has the theorem prover seen much use outside SRI?

Dr. Moore: Both Ford Aerospace and Honeywell use it. I don't know how to quantify success, but there is a lot of interest. A certain amount of skill is needed to use the theorem prover, so we have run courses to teach people what is necessary. A naive user would fail to get a solution from the system where I could succeed, because I would be able to reformulate a lemma.

Professor Randell: The most useful thing the system could say when it fails to find a proof is why.

Dr. Moore: When the theorem prover fails it stops with the formula that failed. One can then construct a counter-example from which it is possible to generate the input data which causes the program to fail.

Professor Katzenelson: Does the verification depend on the size of the program?

Dr. Moore: Program verification is very dependent on the size of the program. It is difficult to specify large programs and the equations you get out are very large.

References

1. American National Standards Institute, Inc., American National Standard Programming Language FORTRAN, ANSI X3.9-1978, 1430 Broadway, New York, New York 10018, April 3, 1978.

2. W.W. Bledsoe, "Splitting and Reduction Heuristics in Automatic Theorem-proving", Artificial Intelligence, 3, pp. 27-60 (1972).

3. W.W. Bledsoe, R.S. Boyer and W.H. Henneman, "Computer Proofs of Limit Theorems", Artif. Intell., 3 (1972) pp. 27-60.

4. W.W. Bledsoe, "A New Method for Proving Certain Presburger Formulas", Advance Papers, 4th Int. Joint Conf. on Artif. Intell., Tbilisi, Georgia, U.S.S.R. pp. 15-21, (September 1975).

5. R.S. Boyer, J.S. Moore and R.E. Shostak, "Primitive Recursive Program Transformation", Proc. Third ACM Symposium on the Principles of Programming Languages", pp. 171-174, Atlanta, Georgia, ACM, New York, New York (1976).

6. R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm", Commun. Assoc. Comput. Mach., 20(10), pp. 762-772 (1977).

7. R.S. Boyer and J.S. Moore, A Computational Logic, Academic Press, New York, (1979).

8. R.S. Boyer and J.S. Moore, "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures", in The Correctness Problem in Computer Science (eds. R.S. Boyer and J.S. Moore) Academic Press, London (to appear, 1981).

9. R.S. Boyer and J.S. Moore, "A Verification Condition Generator for FORTRAN", in The Correctness Problem in Computer Science (eds. R.S. Boyer and J.S. Moore) Academic Press, London (to appear, 1981).

10. C. Chang and R.C.T. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press (1973).

11. A. Church, Introduction to Mathematical Logic, Vol. I, Princeton, Princeton University Press (1956).

12. A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel, "Un Systeme de Communication Homme-Machine en Francais", Groupe de Researche en Intelligence Artificielle, Universite d'Aix-Marseille, Luminy (1972)

13. E. Feigenbaum and J. Feldman, Computers and Thought, McGraw-Hill Book Company, New York, (1963).

14. R. Floyd, "Assigning Meanings to Programs", Mathematical Aspects of Computer Science, Proc. Symp. Appl. Math., Vol. XIX, pp. 19-32, American Mathematical Society, Providence, Rhode Island (1967).

15. P.Y. Gloess, "An Experiment with the Boyer-Moore Theorem Prover: a Proof of the Correctness of a Simple Parser of Expressions", Proc. 5th Conf. on Automated Deduction, Lecture Notes in Computer Science, Vol. 87, pp. 154-169, Springer-Verlag (1980).

16. M. Gordon, R. Milner and C. Wadsworth, "Edinburgh LCF", Computer Science Department, Edinburgh University, CSR-11-77, (1977).

17. M. Gordon, The Denotational Description of Programming Languages, Springer-Verlag, New York, New York (1979).

18. C. Hoare, "An Axiomatic Basis for Computer Programming", Commun. Assoc. Comput. Mach., 12(10), pp. 576-583 (1969).

19. G. Huet and D.C. Oppen, "Equations and Rewrite Rules, A Survey", CSL Technical Report, SRI International, Menlo Park, Ca. U.S.A.

20. L.S. Jutting, "Checking Landau's 'Grundlagen' in the AUTOMATH System", Ph.D. Thesis, Eindhoven University of Technology (1976).

21. R. Kowalski, "Predicate Calculus as a Programming Language", Information Processing 74, North-Holland Publishing (1974).

22. R. Kowalski, "A Proof Procedure Using Connection Graphs", J. Assoc. Comput. Mach., 22, pp. 572-595 (1975).

23. S. Litvintchouk and V. Pratt, "A Proof-Checker for Dynamic Logic", proceedings of the Fifth Int. Joint Conf. on Artif. Intell., Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, U.S.A. (1977).

24. D.W. Loveland, Automated Theorem Proving: A Logical Basis, North Holland Publishing Co. (1978)

25. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", Commun. Assoc. Comput. Mach., 3(4), pp. 184-195 (1960).

26. J.S. Moore, "A Mechanical Proof of the Termination of Takeuchi's Function", Information Processing Letters, Vol. 9, No. 4, pp. 176-181 (1979).

27. J.S. Moore, "Introducing Iteration into the Pure LISP Theorem Prover", IEEE Trans. Software Eng. 1(3), pp. 328-338 (1975).

28. J. Moses, "Algebraic Simplification: A Guide for the Perplexed", Proc. 2nd ACM Symposium on Symbolic and Algebraic Manipulation, (ed. S.R. Petrick) (1971).

29. D. Oppen, "Reasoning about Recursively Defined Data Structures", CS Report STAN-CS-78-678, Stanford University (1978).

30. J.A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle", J. Assoc. Compt. Mach. 12, pp. 23-41 (January, 1965).

31. J.A. Robinson, Logic: Form and Function, North Holland Publishing Company, New York (1979).

32. P. Roussel, "PROLOG: Manuel de reference et d'utilisation", Groupe de Researche en Intelligence Artificialle, Universite d'Aix-Marseille, Luminy (1975).

33. R. Shostak, "Deciding Linear Inequalities by Computing Loop Residues", CSL Technical Report, SRI International, Menlo Park, Ca. U.S.A. (1978).

34. T. Skolem, "On Mathematical Logic", in From Frege to Goedel, (ed. J. van Heijenoort) Harvard Univ. Press, Cambridge, Massachusetts (1967).

35. United States of America Standards Institute, USA Standard FORTRAN, USAS X3.9-1966, 10 Eash 40th Street, New York, New York 10016, March 7, 1966.

36. H. Wang, "Towards Mechanical Mathematics", IBM J. Res. Develop. 4, pp. 2-22 (January 1960).

37. D. Warren, "Implementing PROLOG, a Language for Programming in Logic", Department of Artificial Intelligence, University of Edinburgh (1976).

38. R.W. Weyhrauch, "A User's Manual for FOL", Computer Science Department, Stanford University, STAN-CS-77-432 (1977).