Stanford Artificial Intelligence Laboratory
Memo AIM-256

December 1974

Computer Science Department
Report No. STAN-CS-74-474

Automatic Program Verification III:

A METHODOLOGY FOR VERIFYING PROGRAMS

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-256

DECEMBER 1974

Computer Science Department
Report No. STAN-CS-74-474

AUTOMATIC PROGRAM VERIFICATION III:

A METHODOLOGY FOR VERIFYING PROGRAMS

by

F.W.v.HENKE and D.C. **LUCKHAM**

ABSTRACT

The paper investigates methods for applying an on-line interactive verification system designed to prove properties of PASCAL programs. The methodology is intended to provide techniques for developing a debugged and verified version starting from a program, that (a) is possibly unfinished in some respects, (b) may not satisfy the given specifications, e.g., may contain bugs, (c) may have incomplete documentation, (d) may be written In non-standard ways, e.g., may depend on user- defined data structures.

The methodology involves (i) interactive application of a verification condition generator, an algebraic simplifier and a theorem-prover; (ii) techniques for describing data structures, type constraints, and properties of programs and subprograms (i.e. lower level procedures); (iii) the use of (abstract) data types in structuring programs and proofs.

Within each unit (i.e. segment of a problem), the interactive use is aimed at reducing verification conditions to manageable proportions so that the non-trivial factors may be analysed. Analysis of verification conditions attempts to localize errors in the program logic, to extend assertions inside the program, to spotlight additional assumptions on program subfunctions (beyond those already specified by the programmer), and to generate appropriate lemmas that allow a verification to be completed. Methods for structuring correctness proofs are discussed that are similar to those of "structured programming".

A detailed case study of a pattern matching algorithm illustrating the various aspects of the methodology (including the role played by the user) is given.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 . The views and conclusions contained in this document nre those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U.S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 2215 1.

A METHODOLOGY FOR VERIFYING PROGRAMS

1. INTRODUCTION

We are concerned here with the question of whether or not program verification systems that are currently being developed have any practical usefulness. Verifications of simple standard programs have been obtained with these systems (See for example, [King and Floyd], [Igarashi,London and Luckham], [Deutsch], [Good and Ragland], [Elspas, Levitt,and Waldinger], [Suzuki], [Boyer and Moore]). These results provide encouragement to explore further. But, in all cases except for one example in [Morales] the programs were known in advance to be correct – i.e. provably consistent with their documentation. Moreover, these example test programs are based on standard well-known functions and data structures (for the most part, either integer arithmetic or very simple list processing). Realistically practical verification problems have yet to be faced. A methodology for using these systems to construct verifications in real life situations has not been developed, and indeed the question of whether they will help the process of writing and verifying programs or will merely "get in the way" is entirely open.

The goal of practical usefulness does not imply that the verification of a program must be made independent of creative effort on the part of the programmer. As we shall see later, such a requirement is utterly unrealistic. What we have to do is to provide a tool (the verification system) and instructions for its use (the methodology) that can sometimes enable a programmer to gain a degree of certainty about his or other people's programs. The tool and methods must be easy to apply. In short, we seek to extend the programmer's repertoire of techniques, not to replace it.

The verification system discussed here has been developed specifically for programs written in PASCAL [Wirth] and Is an extension (see [Suzuki]) of the system described in [ILL]. The purpose of this system is to aid the programmer in constructing a proof that his program satisfies its documentation. Such a proof (in the logic of programs [Hoare71,ILL]) is called a verification of the program. The documentation may include:

1. input-output specifications,
2. properties of certain crucial internal states,
3. specifications and properties of sub-programs,
4. specifications of data structures.

In order to be useful in practice we must develop a methodology for using the verifier which aids the user in situations where:

1. the documentation is incomplete (i.e. additional facts about the program must be discovered before a verification can be found),
2. the program itself is unfinished (e.g. parts of it may be unwritten),
3. the program is badly written (even though it conforms to structuring principles),
4. the data structures are non standard (e.g. an axiomatic description does not already exist).

What "aid" should one expect from a verification system? A verification proof depends upon a set of assumptions or lemmas about components of the program (sub-procedures, data structures, library routines, etc.). Let us call this a BASIS for a verification. Essentially, a verification basis is a set of consequences of an underlying axiomatitation of the data structures and subroutines, although such an axiomatization may not actually be known. Different proofs have different bases.

A verification is convincing to a programmer only if he "believes" the basis in the rather imprecise sense that its statements seem true; a more precise sense (acceptable) is given below. As we shall show in examples, a programmer can, obtain a verification of his program using a verifier, and be faced with an impressively complex basis, (or even worse, with some systems he might end up without knowing the basis at ail). If he does not believe the basis, he must be able to reduce its elements to more believable statements or else search for an alternative basis, Thus verification methodology must

1. establish that a basis is adequate, (i.e. ensure the existence of a corrrectness proof from the basis),
2. present alternative bases to the programmer, (i.e. help him discover bases and improve documentation),
3. include methods for analyzing a basis and reducing its components to other bases.

There is an underlying motivational assumption here: in dealing with real life problems it may often be unrealistic and impractical to attempt a verification directly from first principles. It is sufficient to establish a verification basis that is clearly implied by an axiomatic semantics for those concepts that are used in the program.

However, in the case of a "new" program such semantics may not have been formulated. Consequently, we need a methodology which permits a verification to proceed by developing a hierarchy of bases in which a basis at one level verifies elements of the basis immediately above it and depends on bases at the level immediately below. The development of this hierarchy can be viewed as "discovering" the semantics; it will usually be guided by the structure of the program. A basis for verifying properties of one level of the program will be formulated in terms of concepts used in writing that level. The statements in the lowest level bases should be already established facts (about nonprimitives) or axioms for the semantics of primitive functions and data structures. Apart from the practical need to divide complicated verifications into subproofs which may be attempted individually, this hierarchical idea has other advantages. It allows a verification to proceed hand-in-hand with the writing of the program. A basis is to be viewed as more than just a set of assumptions for a verification. Often it includes additional necessary properties of unwritten subroutines beyond what was in their original specifications. Alternatively, the omission of a specification might indicate that a simpler subroutine will suffice. Thus, a basis for one level of a program is a sufficient set of specifications for the next level. Secondly, if an axiomatic semantics for new concepts is needed, it is probably best developed from a knowledge of adequate verification bases (consisting of simple statements) for programs using those concepts. Thirdly, the problem of getting differing programmers to agree upon a "verification" of a program can be terminated short of a complete reduction of the problem to first principles if they both have confidence in the acceptability of some intermediate basis.

At this point we can be a little more precise about some of the concepts we have introduced:

A set of statements forms a basis for verifying a property of a program if a proof of that -property can be given within the logic of programs [Hoare71,ILL] which assumes (i.e. depends upon) only those statements. For emphasis, we shall sometimes say that such a basis is adequate.

A basis is acceptable if (i) all of its statements about the primitives (data structures and library routines) are true, and (ii) programs can be constructed to satisfy all of those statements that contain names for uncoded subroutines.

The primary problem is to find acceptable verification bases. There are a number of important secondary problems. These can all be categorized as parts of the "Formalization problem". First, there is the question of what documentation to include with the program; for example which internal states need to be described, which invariant properties of a loop need to be stated, and what properties of subroutines are actually necessary. Secondly, how should the documentation be expressed? This involves the choice of representation of concepts (e.g. should the relation "<" on the integers be used or can all the necessary facts be expressed in terms of a derived concept like "IS ORDERED SET"?). Also the programmer must choose whether to express internal properties of the program by purely "static" assertions about the values of its variables, or by defining extra computations and making assertions about new variables (i.e. the technique of introducing "ghost" variables and "virtual" program [Clint]). Thirdly, how should the program be written in order to make its verification possible. Recent developments in programming language design, pretty much resulting from experience with the debugging problem, such as block structure and restrictions on procedure parameters and global variables, all certainly help. However, many other details in a program influence its verification (e.g. the form of data structure definitions should indicate clearly the assumptions that can be made about the structures), At the moment, these secondary problems are areas where the programmer's ingenuity must be applied. It is to be hoped that verification methodology will eventually develop some relevant guidelines for attacking the formalization problem.

Our methodology can be very roughly outlined as follows. A program level, which may contain calls to uncoded lower level subroutines, is submitted together with some documentation to the verification system. The general methodology divides activity into three phases: debugging the code, constructing inductive assertions, and constructing a basis . At each of these phases the system is used to indicate modifications and changes by means of a methodology depending on analysis of verification conditions (see Section 2.4). (Eventually we intend to incorporate other techniques for analysing programs.) Modified problems are resubmitted for further analysis, In the third phase the system provrdes a test for the adequacy of a proposed basis. Finally, the basis must be shown to be acceptable, which involves writing the next level of the program.

We shall show in Section 3 how the Pascal Verifier can be used interactively to verify levels in a program as they are written and to guide writing subsequent levels. We illustrate the methodology in action in an experiment to write and verify a program for a fundamental pattern matching algorithm (Unification). We have tried to keep our presentation as close to the real life sequence of events as possible without too much repetition. Essentially, we present snapshots of this sequence of events, each snapshot illustrating a different situation which the methodology must handle. There are examples of the use of the verifier to find bugs, to augment documentation, to build up a basis, and to analyze the basis (i.e. reduce it to simpler statements). This last point involves choosing a formalism for defining recursive data structures, and here we have adopted with minor modifications some suggestions of [Hoare 73]. Of course, our methodology is far from complete, and many of the problems that arise during a verification, (except for the adequacy of a basis, which is handled automatically by the system) involve the user in making choices and decisions. It is already clear how to automate some of this work. However, we must emphasize that the verifier is intended for use in conjunction with other programming facilities .

Some parts of the general methodology depend on a knowledge of what the components of the verifier do. We have, therefore, included a brief description of the verifier in Section 2 together with a simple example of its use.

The principle references upon which this paper depends are [Hoare 71] and [ILL] (for the logic programs), [Hoare and Wirth] (for axiomatic semantics of Pascal), and [ILL] and [Suzuki] (for details of the verifier).We shall use concepts and notation from [Hoare 71, ILL] without definition.

2. THE VERIFIER

The Pascal verification system is represented in outline in Figure 1. The logical theory and implementation of the Verification Condition Generator (VCG) is given in [ILL], and details of the simplifier are in [Suzuki]. In section 3 we shall describe interactive use of this system that relies mainly on these two components and, at the moment, only employs the theorem prover when everything else fails. Here we give a very brief sketch of VCG and the simplifier with the intention of mentioning just those details that affect the methodology of Section 3.

```
   INPUT          --w--m---     --------------         ----------
  PROGRAM         | VCG         |SIMPLIFIER|           | THEOREM |
    and   ------>|         |--->|               |---->| PROVER |
DOCUMENTATION     --w--w---     ---------------        ----------
      ^                              |                      |
      |                              |                      |
      |                              |                      |
      |                              v                      |
      |        MOOIFIEO            --------------------     |
 ---------------<----------|ANALYSIS OF OUTPUT|<---|
              PROBLEM              --------------------
```

Figure I: Main Components of the Verifier

2.1 VERIFICATION CONDITION GENERATOR (VCG)

The input to VCG is a verification problem of the form P{A}Q where P and Qare entry and exit specifications (called assertions) for a Pascal program A. The program A may itself contain additional documentation. Figure 2 shows an input to VCC together with some extra documentation (explained later). To verify that A satisfies its specifications, we require that a proof of P{A}Q within the logic of programs be found. VCG reduces problems of the form P{A}Q to problems about shorter programs, using the rules of the axiomatic semantics of Pascal. For example, P{IF L THEN B ELSE C}Q could be reduced to verifying two problems, $P\wedge L\{B\}Q$ and $P\wedge\neg L\{C\}Q$; the axiomatic semantics for conditional statements implies that if these latter two problems are verified then the first problem is also verified. Similar reductions are applied to other kinds of Pascal statements. The final output from VCG is a set of purely logical statements composed from Pascal Boolean assertions (see Figure 3). These are called the Verification Conditions (abbreviated to VC's) for the original problem, P{A}Q.

There are two points to be mentioned here. First of all, VCG has a completeness property with respect to provability. Assume that a verification of P{A}Q is to be found making only assumptions from some underlying axiomatic semantics, T say. A proof of P{A}Q can always be constructed assuming the VC's, and conversely if P{A}Q is provable in the logic of programs from T, then VCG will generate VC's that are provable in T provided A contains additional helpful assertions (exactly what extra documentation must be given is a subject of much current research). This means that the set of VC's is always an adequate Basis for the verification (but it may not be acceptable). And also, if the user's problem is provable from statements in T, he will be able to establish that fact with the present verifier by adding enough documentation to the program. The second point is that VCC reduces problems to purely logical VC's. As we shall see later, this may not always be the best strategy, especially when the VC's involve the names of procedures that have yet to be written, and it may sometimes be better to stop the reduction process and generate VC's that contain pieces of code explicitly. It is doubtful if verification can be based solely on pure logic, and it may be necessary to use other techniques such as equivalence preserving transformations on programs.

Finally, the present version of VCG contains a number of new features and rules that are not in the original version in [ILL]. The one most relevant to our discussion is a feature (due to Suzuki) for handling calls to uncoded functions by means of "DEFFUN" statements. The intention is to give the user an easy way to state specifications for functions that are not yet coded, although it can be used for standard functions as well. A DEFFUN statement is of the form:

```
DEFFUN     f(x1:type1,...): type
ENTRY      R(x 1,...); EXIT <Value>:S(f);
```

where "f" is the function name, <value> is an expression denoting the value of f, and R(x 1,..) and S(x 1,...) are entry and exit assertions. No function body is required. Whenever a call to f occurs during the generation of VC's the adaptation rule [Hoare] will be applied:

$$\frac{P\{A\}(R(a,\ldots) \wedge \forall(a',\ldots)(S(f(a',\ldots))\rightarrow Q(f(a',\ldots))))}{P\{A;x\leftarrow f(a,\ldots)\}Q(x)}$$

A verification of the program will then imply the runtime legality of all calls to f. The use of DEFFUN'S is not mandatory, and the user may choose to omit them if he is sure that all his function calls are legal (a normal compile-time type check may be sufficient).

2.2 THE SIMPLIFIER

Many VC's are (or contain subformulas that are) lengthy and complicated but turn out to be logically trivial. The first step in the analysis of VC's is to simplify and eliminate the trivial parts so that one can see the real verification problems. It is inappropriate to process these unsimplified VC's with the theorem prover because there are faster, less general techniques for carrying out logical and algebraic formula reduction. VC's are first processed by a simplifier. Originally, we had planned the simplifier as a preprocessor to the theorem prover, but our current methodology makes repeated interactive use of the simplifier before using the prover (See Figure 1).

Let us first state very briefly what the simplifier does (Full details in [Suzuki]). The user may submit three kinds of documentation statements which will be used as reduction rules by the simplifier. Here are examples of each:

AXIOM CAR(CONS(@X,@Y))↦X;

This means that any term in a VC that "matches" the left side (i.e. is identical to the left side when X and Y are replaced by appropriate strings) will be replaced in the VC by the string for X. It is a left to right reduction rule. A variable preceded by "@" is called a pattern variable.

AXIOM IF ISTERMLIST(L)∧¬(L=ZERO) THEN ISTERM(HD(@L))↦TRUE

This is a conditional axiom. Suppose a VC has the form A→B. Any expression in B that matches ISTERM(HD(@L)) may be reduced by this rule to TRUE if ISTERMLIST(L) and ¬(L=ZERO) (where L is the substitution string for @L in the successful match) occur in A.

GOAL RCONS(@X1,@Y1)=RCONS(@X2,@Y2) SUB (X1=X2)∧(Y1=Y2)

This is a goal statement. It is treated as a reduction rule that says "an expression that matches the GOAL may be replaced by TRUE if the corresponding instance of the SUBgoal can be reduced to TRUE.

Figure 2 shows a program with documentation that will be used as simplification rules.

Goal statements can be formulated as conditional axioms and vice versa. The difference is that axioms are "sticky" (any reduction by an axiom is never reversed) whereas goals are not (goals have no effect on a VC unless the reduction can be pushed all the way to TRUE). Ideally, the axioms should consist of those reduction rules having the property that no reduction to TRUE depends on their order of application.

The simplifier contains a sequence of simplifying "boxes", An incoming VC is simplified in sequence by (l) a logical proposition simplifier, (2) processing of arithmetical expressions by choice of standard forms and by evaluation, (3) reduction by axioms, and (4) reduction by goals.

This is a good place to discuss the role of the simplifier in our verification methodology. Essentially, we are using the simplifier as a fast theorem prover. Our philosophy is that the user should be able to submit a problem and receive back the reduced VC's within a few seconds. If the kinds of reduction rules are easily understood, he will probably be able to see further useful rules by analyzing the VC's. He can then resubmit the problem with additional rules. Eventually some of this analysis will be automated (See Section 3) and likely rules suggested to the user. There is no attempt to make the set of rules logically independant at first, the idea being to develop a first basis quickly. It does make sense to choose simple rules(believability), and some kinds of rules (e.g. commutativity) have to be excluded because of the way the simplifer works. If all VC's reduce to TRUE, the set of reduction rules is an adequate verification basis.

The kinds of reduction rules have to be simple also for speed as well as understandability.

However, experience suggests that we do need something beyond algebraic manipulation. The goal statements form a simple theorem prover. On the other hand, some complex propositional transformations are time consuming and often-unneeded, and best left to the theorem prover. Thus the boarderline between Simplification and Theorem-proving, at the moment, is somewhat pragmatic.

2.3 AN EXAMPLE Figure 2 shows the procedure SIFTUP used in the algorithm TREESORT3 [Floyd] for sorting linear arrays of integers. The problem is to verify that the output of SIFTUP is always a permutation of its input. The program contains an internal ASSERTION as **well** as the entry and exit conditions for this problem.

There are three reduction rules stated in terms of the relation PERMUTATION (A,B) meaning "array A is a permuation of array B", and the function ASET(A,i,j) which applies A[i]←j to A. We may have no specific axiomatic theory of permutations in mind. Nevertheless, the first two AXIOMS are clearly trivial. Most people will "believe" the third one after a moments thought.

The unsimplified VC's put out by VCC are in Figure 3. So also are the simplified ones, from which we conclude that the three rules are an adequate basis for verifying the permutation property. The reader may wonder how we thought of the third rule. What we did was to run the problem first without it and compare the premiss and conclusion of VC•3 or •4.

```
AXIOM PERMUTATION(@I,@I)↔TRUE;
AXIOM ASET(@I1,@I2,@I1[@I2])↔I1;
AXIOM PERMUTATION(ASET(ASET(@I1,@I2,@I1[@I3]),@I3,@I4),@I5)↔
      PERMUTATION(ASET(I1,I2,I4),I5);

PROCEDURE SIFTUP(IO,N:INTEGER);
ENTRY M=MO;
EXIT  PERMUTATION(M,MO);
VAR COPY:REAL; J, I:INTEGER;
BEGIN
      I ← IO; COPY ← M[I];
   10: J ← 2 * I;
      ASSERT PERMUTATION(ASET(M,I,COPY),MO);
      IF J ≤ N THEN
     . BEGIN IF J < N THEN
             BEGIN IF M[J+1]>M[J] THEN J ← J+1 END;
      IF M[J]> COPY THEN BEGIN M[I]←M[J]; I ← J; GO TO 10    END;
      END;
      M[I]← COPY;
END;
```

Figure 2: The procedure **SIFTUP used** by TREESORT.

```
# 1
M=MO → PERMUTATION(ASET(M,IO,M[IO ]),MO)
# 2
(COPY<M[J+1])∧(M[J]<M[J+1])∧(J<N)∧(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO)
     → PERMUTATION(ASET(ASET(M,I,M[J+1]),J+1,COPY),MO)
# 3
(COPY<M[J])∧¬(M[J]<M[J+1])∧(J<N)∧(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO)
     → PERMUTATION(ASET(ASET(M,I,M[J]),J,COPY),MO)
# 4
(COPY<M[J])∧¬(J<N)∧(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO)
     → PERMUTATION(ASET(ASET(M,I,M[J]),J,COPY),MO)
# 5
¬(COPY<M[J+1])∧(M[J]<M[J+1])∧(J<N)∧(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO)
     → PERMUTATION(ASET(M,I,COPY),MO)
# 6
¬(COPY<M[J])∧¬(M[J]<M[J+1])∧(J<N)∧(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO)
     → PERMUTATION(ASET(M,I,COPY),MO)
# 7
¬(COPY<M[J])∧¬(J<N)∧(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO)
     → PERMUTATION(ASET(M,I,COPY),MO)
# 8
¬(J≤N)∧PERMUTATION(ASET(M,I,COPY),MO) 4 PERMUTATION(ASET(M,I,COPY),MO)


THE SIMPLIFIED VERIFICATION CONDITIONS ARE:

# 1  TRUE
# 2  TRUE
# 3  TRUE
# 4  TRUE
# 5  TRUE
# 6  TRUE
• 7  TRUE
# 8  TRUE

TIME: 7 CPU SECS, 31 REAL SECS
```

Figure 3: VERIFICATION CONDITIONS FOR **SIFTUP**

2.4 HINTS ON ANALYSING VC'S. Each VC corresponds to a path through the program between two assertions (possibly the same assertion). A simple VC has the form $P \rightarrow Q\alpha$ where Q is the end assertion, P is a logical combination of the beginning assertion and boolean control tests, and α is a substitution of terms for program variables. If the path contains function or procedure calls, the form of the VC is more complex. The VC expresses a logical condition on the action of the program along the path. It also contains implicitly a description of the path and what the action is.

(a) The path of a VC is determined by the values of the boolean control tests occuring in P.

(b) The computational changes can be determined from terms substituted for program variables by α.

EXAMPLE: VC4 (figure 3) corresponds to the path from the ASSERTION satisfying $J<N$, $\neg(J<N)$, and $M[J]>COPY$ back to the ASSERTION. The action of α (determined from the Q-part of VC4) is: $M \leftarrow ASET(M,I,M[J])$ (i.e. $M[I] \leftarrow M[J]$), and $I \leftarrow J$. The assignment $J \leftarrow 2*I$ cannot be detected unless ASSERTION contains J.

Our methodology depends on extracting information from VC's. When a VC does not reduce to TRUE, the programmer may try to decide if it is true using his knowledge of the program (i.e. the path and action). If it is true, he can either expand P (i.e. the beginning assertion) or give additional documentation in order to prove the VC. Additional documentation can be given by placing new assumptions in the basis. If the VC appears to be false, he has either to weaken the specifications (changing P or Q) or to change the program .

Commonly occuring situations include the following:

(i). Paths of VC's correspond with cases the program is supposed to recognize. Any kind of mismatch of cases and paths Indicates a change should be made in the program.

(ii). The action of a VC does not express what the program was intended to do in the case corresponding to the VC path. A change in the program is necessary (see 3.1 (b),(c)),

(iii).Part of Q is logically independant of P. Then usually P should be expanded (see 3.1(a) and (d)).

(iv). The VC appears true but not provable from the the current Basis. Analysis of the components of Q and related parts of P can often yield conditions on functions and procedures which were overlooked or were omitted because their relevence was questioned. These may then be added to the Basis (see 3.3 a,b,c).

How much of this analysis and corrective action can be automated ? Most of the current attempts to automate the construction of assertions (especially in case (iii)) assume that the program is already correct. If we do not assume correctness, it seems that the choice of action (whether to change the documentation or the program) depends entirely on the programmer's intentions and cannot be automated. However, much can be done to automate the extraction of information from VC's. The system helps by displaying the (updated) effects of any changes and allowing experimentation.

3. CASE STUDY: METHODOLOGY IN ACTION

Let us first explain unification informally A unification program accepts as Input two lists of terms X,Y and constructs as output a substitution (of terms for variables) that makes each member of X equal to the corresponding member of Y if possible, or else outputs the answer "IMPOSSIBLE". For example, given the input $X=\{x,g(h(y),z),v\}$, $Y=\{f,g(h(u),k(w)),v\}$, the program should put out the unifier $Z=\{<x \leftarrow f>, <y \leftarrow u>, <z \leftarrow k(w)>\}$.

It is possible to write such first-order unification programs in many different ways. A popular method is to use the input list structures as temporary storage, which permits an encoding as an iterative loop without recursive calls. If this is done in the example above, a first pass through the loop will result in the values $X'=\{g(h(y),z),v\}$, $Y'=\{g(h(u),k(w)),v\}$, $Z'=\{<x \leftarrow f>\}$; a second pass will yield $X''=\{v,h(y),z\}$, $Y''=\{v,h(u),k(w)\}$, $Z''=\{<x \leftarrow f>\}$, and so on.

We asked an experienced programmer to write a unification program in real time (while we looked on). We note the following. He stated his intention to use the input data structures as temporary storage, but no structures were declared. He attempted to code "top down", naming subfunctions without coding them, but merely stating what they were supposed to do (but he often changed his mind). As the program developed, he had difficulty documenting the loop and introduced virtual program to do this (without telling us). He gave up on the idea of purely iterative code and ended up putting a recursive call inside a WHILE loop.

It is this program that we start with as VERSION I. We make no claim that it is how a unification program should be coded. We choose it because it is the result of a real life situation and is a problem of sufficient richness to be a good test of our ideas on methodology.

The property to be verified is that if the program stops, either it outputs a unifier Z of the input termlists X and Y or it outputs a failure. Other standard (and complementary) properties that will be verified later are that Z is a most general unifier, and that if a failure is output then X and Y are not unifiable.

The top level program is developed in three steps: version 1 (a first sketch), a debugged version 2, and the final verified version. Ideally, debugging and verification happen simultaneously; for demonstration purposes, we largely separated these two steps. As it is not our aim to discuss syntactic analysis, all programs are given in a syntactic correct form which will be accepted by the VCG-system (or a compiler).

The subfunctions used in the program have the following intended meanings:

TSUBST(X,Z)	- the term resulting from applying substitution Z to term X,
SUBST(X,Z)	- the termlist resulting from applying substitution Z to termlist X,
ZERO	- the empty list,
COMP(Z,X,Y)	- the substitution resulting from composing substitution Z with the single substrtution that replaces variable X by term Y,
OCCUR(X,Y)	- a Boolean test,TRUE whenever term X is a subterm of term Y,
RCONS(U,X)	- termlist obtained by adding term X to the end of termlist U,
TERMS(X)	- the termlist consist of the arguments of term X (not a simple variable),
FNLT(X)	- the function letter of complex term X,
HD(X), TL(X)	- the head and tail of list X.

3.1 VERSION 1: DEBUGGING AND EXTENDING DOCUMENTATION. Version 1 is the top level of the program that was initially submitted for verification. It was written almost on-line and therefore contains bugs and even misconceptions of the structure of algorithm and data. Roughly speaking it is a sketch of a program with the question "can this be made to work?" It does not include any specifications of the data types (in form of axioms, deffuns etc.). The invariant of the main loop consists just of the main idea: the initial parts of the termlists X and Y are unified by the constructed substitution Z. To express this the programmer used two "ghost variables" [Clint] U and V, which hold the parts already dealt with, and "virtual program", i.e., statments that are not necessary for the actual computation. Failure of the algorithm is expressed by the pseudo-procedure LOSE.

Note that the program contains several bugs:

- the cases structure is incorrect;
- after the recursive call of UNIFY the result is not tested for success or failure and the returned substitution is not assigned to Z;
- at the end of the procedure it is not guaranteed that both Xl and Y l are ZERO.

```
PASCAL
PROCEDURE UNIFY(X,Y:TERMLIST;Z1:SUB; VAR Z:SUB);
  ENTRY ISTERMLIST(X)∧ISTERMLIST(Y);
  EXIT (SUBST(X,Z)=SUBST(Y,Z)) v LOSE(X,Y);

VAR U,V,X1,Y1:TERMLIST; VAR X2,Y2:TERM; VAR Z2:SUB;
BEGIN

% Initialization of variables %
U:=ZERO;V:=ZERO;Z:=Z1; X1:=X; Y1:=Y;

INVARIANT (SUBST(U,Z)=SUBST(V,Z)) v LOSE(X,Y)

WHILE (X1≠ZERO)∧(Y1≠ZERO) DO
  BEGIN
  X2:= SUBST(HD(X1),Z);
  Y2:= SUBST(HD(Y1),Z);
  IF ISVAR(X2) THEN     BEGIN IF ISVAR(Y2) THEN Z:=COMP(Z,X2,Y2);
                              IF OCCUR(X2,Y2) THEN LOSE(X,Y)
                                ELSE Z:=COMP(Z,X2,Y2)
                        END
   ELSE BEGIN IF ISVAR(Y2)
                  THEN BEGIN IF OCCUR(Y2,X2) THEN LOSE(X,Y)
                                 ELSE Z:=COMP(Z,Y2,X2)
                        END
                 ELSE BEGIN IF FNLT(X2)=FNLT(Y2)
                                 THEN UNIFY (TERMS (X2),TERMS (Y2), Z,Z2)
                                ELSE LOSE(X,Y)
                        END
          END;
   U :=RCONS(U,HD(X1)); V :=RCONS(V,HD(Y1));
   X1:=TL(X1);  Y1:=TL(Y1);
   END; % End of WHILE body %

END; % Procedure body %
```

Figure 4: Version1

```
LOSE NOT FOUND

* 1
(ISTERMLIST(Y) & ISTERMLIST(X) & LOSE(X,Y) v SUBST(U*1,Z*1)=SUBST(V*1,Z*1) &
 ¬¬Y1*1=ZERO A ¬X1*1=ZERO
→LOSE(X,Y) v SUBST(Y,Z*1)=SUBST(X,Z* 1))

* 3
(LOSE(X,Y) v SUBST(U,Z)=SUBST(V,Z) & ¬Y1=ZERO & ¬X 1 =ZERO &
 ¬ISVAR(SUBST(HD(X1),Z)) &
 ¬ISVAR(SUBST(HD(Y 1 ),Z)) & FNLT(SUBST(HD(Y 1),Z))=FNLT(SUBST(HD(X1),Z))
→ ISTERMLIST(TERMS(SUBST(HD(Y1),Z))) &
  (LOSE(TERMS(SUBST(HD(X1),Z)),TERMS(SUBST(HD(Y 1 ),Z))) v
  SUBST(TERMS(SUBST(HD(Y1),Z)),Z2* 1)=SUBST(TERMS(SUBST(HD(X 1),Z)),Z2*1)
 →LOSE(X,Y) v SUBST(RCONS(U,HD(X 1 )),Z)=SUBST(RCONS(V,HD(Y 1 )),Z)) &
   ISTERMLIST(TERMS(SUBST(HD(X 1 ),Z))))

* 9
(LOSE(X,Y) v SUBST(U,Z)=SUBST(V,Z) & ¬Y 1 =ZERO & ●  QX 1 =ZERO &
 ISVAR(SUBST(HD(X 1 ),Z)) &
 ISVAR(SUBST(HD(Y 1 ),Z)) & OCCUR(SUBST(HD(X1),Z),SUBST(HD(Y 1 ),Z))
→ PRE_LOSE(X,Y) & (RES_LOSE(X,Y)
 → LOSE(X,Y)  v
    SUBST(RCONS(U,HD(X 1 )),COMP(Z,SUBST(HD(X 1 ),Z),SUBST(HD(Y1),Z)))
    =SUBST(RCONS(V,HD(Y1)),COMP(Z,SUBST(HD(X 1 ),Z),SUBST(HD(Y 1 ),Z)))))
```

Figure 5: Some **VC's for** version **1 in simplified form**

Corresponding to the lack of any detailed information, the system is not able to simplify more than the most trivial parts of the generated VC's.

Discussion of the problems involved in version 1:

a) Failure Handling: Trying to define pre- and post-conditions for the missing procedure LOSE as required by the system, the programmer realizes that indication of failure is a change of the state rather than an action to be invoked (a direct way to put across an error message, e.g. in form of a jump to the top level is not available in Pascal). Thus, he should use a boolean variable FLAG whose value will indicate success or failure. Accordingly, the procedure UNIFY gets one more variable parameter, such that it returns the value of FLAG together with a new value of Z. Each call to LOSE is to be replaced by FLAG:=0, and the initial value of FLAG will be 1. The EXIT assertion must be changed to specify the use of FLAG:

$$\text{EXIT } (\text{SUBST}(X,Z)=\text{SUBST}(Y,Z) \wedge \text{FLAG}=1) \vee \text{FLAG}=0$$

An equal change must be made in the INVARIANT (If the INVARIANT is not changed, the necessity of the change will be seen in later runs in the path leading from the loop to the EXIT.)— see Section 2.4(iii).

b) Missing Code: The necessity to update the value of Z after the recursive call to UNIFY can be detected by analysing VC*3. The relevant parts are

$$\begin{aligned}&\neg\text{ISVAR}(\text{SUBST}(\text{HD}(X\,1),Z)) \,\&\, \neg\text{ISVAR}(\text{SUBST}(\text{HD}(Y\,1),Z)) \,\&\\ &\text{FNLT}(\text{SUBST}(\text{HD}(Y\,1),Z))=\text{FNLT}(\text{SUBST}(\text{HD}(X\,1),Z)) \,\&\, \text{SUBST}(U,Z)=\text{SUBST}(V,Z)\\ &\rightarrow \text{SUBST}(\text{TERMS}(\text{SUBST}(\text{HD}(Y\,1),Z)),Z2*1)=\text{SUBST}(\text{TERMS}(\text{SUBST}(\text{HD}(X\,1),Z)),Z2*1)\\ &\rightarrow \text{SUBST}(\text{RCONS}(U,\text{HD}(X\,1)),Z)=\text{SUBST}(\text{RCONS}(V,\text{HD}(Y\,1)),Z)\end{aligned}$$

VC*3 as it stands is not provable (there are obvious counterexamples). The first two lines indicate that it corresponds to the path containing the procedure call UNIFY(..,Z2). The purpose of this call is to extend Z to a substitution Z2 that unifies the pair HD(X 1) and HD(Y 1) as well as U and V. Indeed, the occurences of Z in the last line of VC*3 should be Z2*1. The final value of Z at the end of the path should be the value of Z2 returned by UNIFY if the attempted unification succeeds. Thus the action on the path is not what was intended, and the code must be changed- Section 2.4(ii). The correct action can be achieved by adding

IF FLAG-1 THEN Z:=Z2;

immediately after the call.

c) Error in the Case Analysis: VC*9 is of the form $P\supset(Q\supset R)$. The programmer notices the combination of Boolean tests

$$\text{ISVAR}(A)\wedge\text{ISVAR}(B)\wedge\text{OCCUR}(A,B)$$

in part P. This means that VC*9 expresses a condition on the action of the program along the path corresponding to this combination of cases. This action can be deduced from Q and R: the "procedure" LOSE is called, and the substitution Z Is updated to COMP(Z,C,D). This combination of actions is clearly wrong; indeed, the programmer's intention in this case is that the program should do nothing to Z and continue—another example of Section 2.4(ii). This error is fixed by adding an extra IF statement for the case $ISVAR(X2) \wedge ISVAR(Y2) \wedge (X2 \neq Y2)$ (see figure 6).

d) Expansion of the INVARIANT: To state the invariant of the loop, the programmer introduced the variables U and V which are intended to hold the values for the initial parts of the termlists X and Y. From looking at VC#1 he can see that

(1) SUBST(U,Z)=SUBST(V,Z)

has to imply

(2) SUBST(X,Z)=SUBST(Y,Z)

when control leaves the loop, i.e. when X1=ZERO and Y1=ZERO, and the algorithm is successful. This is impossible unless some relationship between U,V and X,Y respectively is given — an example of Section 2.4(iii). Now, the intended relationship is

(3) $APPEND(U,X1)=X \wedge APPEND(V,Y1)=Y$

where APPEND is the standard LISP function. The question is, where should this be added to the documentation? Further analysis of VC#1 shows that the only possible place is the invariant of the loop (the other parts of the VC derive from entry and exit condition and the loop control test). The obvious properties of APPEND

(4) APPEND(ZERO,L)=L APPEND(L,ZERO)=L

will be assumed as axioms, guaranteeing that (3) will be true when entering and leaving the loop. Then, (1) will imply (2), provided both X1 and Y1 equal ZERO at the end. On the next run with the two axioms on APPEND added, the omission of a corresponding test after leaving the loop will be visible in the VC, so a statement

IF $(X1 \neq ZERO) \vee (Y1 \neq ZERO)$ THEN FLAG:=&

is added at the end of the procedure.

REMARK The programmer could as well try to figure out what other properties of APPEND are required to prove invariance of the invariant around the loop, but he leaves that to the system as he hopes to find what is needed from the VC's of a subsequent run (refer to Section 3.3 b). Note that the function APPEND is used only in the documentation.

e) Use of Ghost Variables and Virtual Program: As a data flow analysis would show, the variables U and V are not necessary to compute the final result. They are needed only to express the invariant of the main loop. Therefore they are called "ghost variables" [Clint]. Obviously, assignments to ghost variables need not be executed at run time nor translated by a compiler. Thus these statements are considered "virtual"; their purpose is to ensure the correct current values of the ghost variables as the computation proceeds.

The technique of using ghost variables and pieces of virtual program for documentation purposes is very useful and quite common. Although they often could be eliminated as part of the program text - especially in the context of arithmetical problems where most operations are invertible - they represent a powerful tool. Whether a programmer chooses virtual program or not depends on his preferences and the problem domain. In our example, U and V could be replaced in the invariant by expressions like EXCLUDE(X1,X) meaning the remaining initial list after chopping off X1 from the right end of X. In this way, EXCLUDE becomes sort of an inverse function of APPEND. However, we prefer the virtual program approach since it expresses clearer the building-up of the values of U and V simultaneously with the other computation; beside that, the axioms and goals involving the equalities SUBST(U,...)=SUBST(V,...) are complicated even without the difficulties added by the use of EXCLUDE, as will be seen later:

3.2 DATA TYPES AND TYPE CHECKING. For program verification, data type definitions represent sets of axioms defining the semantics of the types. They are primitive statements in the verification basis. This is usually called the "abstract" definition of a data type. A handy formalism is needed that permits the programmer to define his types without having to write down ail the axioms explicitly. The unification program here uses recursive types. We adopt the following formalism for defining recursive data types. It is closely related to suggestions of [McCarthy 1963] and [Hoare 19731, and is an extension of and a departure from what is possible in the present version of Pascal.

A type definition is made by listing alternatives. An alternative is either a simple type (e.g., one that is a type predefined in the language, or a constant) or a composed type. In a more formal BNF-like notation:

```
<type definition>   ← <type name> ':= <type> { | <type> }*
<type>              ← <simple type> | <composed type>
<composed type>     ← <constructor> '( <selector- l>:<type_1>, . . ,<selector_n>:<type_n> ')
                      {'IF <constraint>)
<simple type>       ← <constant> | <type name>
<constraint>        ← <boolean expression of selector names>
```

with the restriction that the names of all constructors in a type definition and all selectors in one composed type have to be distinct. The formal type definition syntax permits simple kinds of constraints to be placed on a constructor.' The meaning of the constraint is that in order to constuct an element of the type, the constuctor must be applied to arguments that satisfy the constraining condition. (an example is the type SINGLESUBstitution below).

In this notation, the data types to be used in our program may be defined by the following (only the upper-case letter part of the names is used in the programs):

```
TERM                 := VAR | MKTERM(FNLT:CONST, TERMS:TERMLIST)
TERMLIST             := ZERO | CONS(HD:TERM, TL:TERMLIST)
SINGLESUBstitution   := PAIR(VAR:VAR,TERM:TERM) IF ¬ OCCUR(VAR,TERM)
SUBstitution         := ZERO | MKSUB(REST:SUB, LAST:SINGLESUB)
```

VARiables and CONSTants are assumed as primitive types. TERMLIST is just a linear list of TERMs. The constraint on SINGLESUB means "Z-PAIR(V,T) is a SINGLESUBstitution only if V does not occur in T."

Notation: IS<typename> denotes the type predicate (i.e. characteristic function) for <typename>.

The type definition determines the logical type of all the functions occuring in it (constructors and selectors). For example, HD maps TERMLIST into TERM, and MKTERM is a function from CONST*TERMLIST into TERM (*: direct product). It is assumed that a selector function is defined only for objects belonging to the corresponding constructed subtype.

At present the verifier does not yet accept type definitions but needs to be given the type axioms. The definition of, e.g., SUBstitution denotes a set of axioms including standard relationships between constructors and selectors:

```
ISSUB(ZERO)
IF ISSUB(A)∧ISSINGLESUB(B) THEN ISSUB(MKSUB(A,B))
REST(MKSUB(A,B))=A
LAST(MKSUB(A,B))=B
```

and the induction rule

$$\frac{F(ZERO) \qquad F(A) \vdash F(MKSUB(A,B))}{ISSUB(S) \vdash F(S)}$$

for any formula F.

The functions defining a type (constructors and selectors) are submitted to the system as DEFFUN's. If there are constraints on a type (as for SINGLESUB), type checking also involves a check if those conditions hold whenever a new object of the type is constructed; thus, the constraints become part of the ENTRY assertion of the DEFFUN for the respective constructors, When the program is augmented by DEFFUN's for all subfunctions, the system will generate complete argument type checks as part of the VC's. However, for reduction of the VC's the assertions have to include a type predicate for each variable that is passed as a parameter to a function or procedure. In this way, the verifier will do type checking automatically.

While formulating the type declarations for the subfunctions it was noticed that in the use of SUBST in the INVARIANT the first argument is a termlist whereas in its function calls in the assignment statements the first argument is a term. In order to avoid this type conflict a separate function TSUBST is introduced for application to terms.

3.3 VERSION 2: CONSTRUCTING A BASIS. Version 2 of the procedure UNIFY (see figure 6 on the nest page) is a correct program in the sense that the code does satisfy the ENTRY/EXIT assertions. The assertions, (including the invariant) have been expanded to a point where they ought to be sufficiently detailed. This version contains those axioms and goals that are naturally anticipated by the programmer. Among those are axioms that express intended properties of the data types and subfunctions. In order to speed up the simplifer, only those data type axioms that were really needed have been added. The DEFFUN's for the basic data type functions have also been included; they consist of just the obvious input and output specifications (essentially type information).

What still remains to be done is to establish an adequate basis for verifying the top level, i.e., completion of the documentation. Below we demonstrate techniques for constructing the basis by extracting from the reduced VC's additional specifications (or "lemmas") on the subfunctions which are believable and which permit the system to completely reduce the VC's to TRUE.

```
GOALFILE        --
% Axioms defining the data types and basic functions %
AXIOM ISTERMLIST(ZERO)↔ TRUE;
AXIOM ISSUB(ZERO)↔TRUE;

% Axioms describing proporties of subfunctions %
AXIOM APPEND(ZERO,@S)↔S;
AXIOM APPEND(@S,ZERO)↔S;
AXIOM SUBST(@X,ZERO)↔X;
AXIOM SUBST(ZERO,@S)↔ZERO;.;

PASCAL

DEFFUN HD(L:TERMLIST):TERM;       ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERM(HD);

DEFFUN TL(L:TERMLIST):TERMLIST; ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERMLIST(TL);

DEFFUN RCONS(L:TERMLIST; X:TERM):TERMLIST;
       ENTRY ISTERMLIST(L)∧ISTERM(X); EXIT ISTERMLIST(RCONS);

DEFFUN TERMS(X:TERM):TERMLIST; ENTRY ISTERM(X)∧¬ISVAR(X);      EXM ISTERMLIST(TERMS);

DEFFUN  FNLT(X:TERM):CONST;       ENTRY ISTERM(X)∧¬ISVAR(X);      EXIT ISCONST(FNLT);

DEFFUN TSUBST(X:TERM;S:SUB):TERM; ENTRY ISTERM(X)∧ISSUB(S); EXIT ISTERM(TSUBST);

DEFFUN SUBST(X:TERMLIST; S:SUB):TERMLIST;
       ENTRY ISTERMLIST(X)∧ISSUB(S); EXIT ISTERMLIST(SUBST);

DEFFUN COMP(S:SUB; X:VAR; Y:TERM):SUB;
       ENTRY ISSUB(S)∧ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y); EXIT ISSUB(COMP);
```

Figure 6 Version 2 (continued)

```
PROCEDURE UNIFY (X,Y:TERMLIST; Z1:SUB; VAR Z:SUB; VAR FLAG:BOOLEAN);
   ENTRY ISTERMLIST(X)∧ISTERMLIST(Y)∧ISSUB(Z1);
   EXIT (ISSUB(Z)∧(SUBST(X,Z)=SUBST(Y,Z))∧(FLAG=1)) ∨ (FLAG =0);

VAR U,V,X1,Y1:TERMLIST; VAR X2,Y2:TERM; VAR Z2:SUB;
BEGIN
% Initialization of variables %
U:=ZERO; V:=ZERO; Z:=Z1; X1:=X; Y1:=Y; FLAG:=1 ;

INVARIANT (ISSUB(Z)∧ISTERMLIST(U)∧ISTERMLIST(V)∧ISTERMLIST(X1)∧ISTERMLIST(Y1)
        ∧(SUBST(U,Z)=SUBST(V,Z))∧(APPEND(U,X1)=X)∧(APPEND(V,Y1)=Y)∧(FLAG=1)) ∨ (FLAG=0)

WHILE (X1≠ZERO) ∧ (Y1≠ZERO) ∧ (FLAG=1) DO
  BEGIN
   X2:= TSUBST(HD(X1),Z);
   Y2:= TSUBST(HD(Y1),Z);
   IF ISVAR(X2) THEN    BEGIN IF ISVAR(Y2)
                              THEN BEGIN IF (X2≠Y2)
                                              THEN Z:=COMP(Z,X2,Y2)
                                         END
                              ELSE BEGIN IF OCCUR (X2,Y2) THEN FLAG:=0
                                                          ELSE Z:=COMP(Z,X2,Y2)
                                         END
                         END
   ELSE BEGIN IF ISVAR(Y2)
                THEN BEGIN IF OCCUR(Y2,X2) THEN FLAG:=0
                                           ELSE Z:=COMP(Z,Y2,X2)
                     END
                 ELSE BEGIN IF FNLT(X2)=FNLT(Y2)
                              THEN BEGIN UNIFY(TERMS(X2),TERMS(Y2),Z,Z2,FLAG);
                                         IF FLAG=1 THEN Z:=Z2
                                   END
                              ELSE FLAG:=0
                      END
         END;

  U :=RCONS(U,HD(X1));   V :=RCONS(V,HD(Y1));
  X1:=TL(X1);            Y1 :=TL(Y1);
  END; % End of WHILE body %

IF (X1≠ZERO) ∨ (Y1≠ZERO) THEN FLAG:=0
END; % Procedure body %
```

Figure 6: **Version 2** (intermediate **version**)

VC's 1 3 5 6 8 are reduced to TRUE

```
* 2
(ISTERMLIST(X) & ISTERMLIST(Y) & ISSUB(Z*1)∧ISTERMLIST(U*1)∧ISTERMLIST(V*1)
 ∧SUBST(U*1,Z*1)=SUBST(V*1,Z*1)∧U*1=X∧V*1=Y∧FLAG*1=1∨FLAG*1=0 & ISSUB(Z1)
→ISSUB(Z*1)∧SUBST(Y,Z*1)=SUBST(X,Z*1)∧FLAG*1=1∨FLAG*1=0)

* 4
(ISTERMLIST(RCONS(V,HD(Y1))) & ISTERMLIST(RCONS(U,HD(X1))) & ISTERMLIST(TL(Y1)) &
 ISTERMLIST(TL(X1)) & ISSUB(Z2*2) &
 SUBST(TERMS(TSUBST(HD(Y1),Z)),Z2*2)=SUBST(TERMS(TSUBST(HD(X1),Z)),Z2*2) &
 ISCONST(FNLT(TSUBST(HD(Y1),Z))) & ISTERMLIST(TERMS(TSUBST(HD(Y1),Z))) &
 ISTERMLIST(TERMS(TSUBST(HD(X1),Z))) &
 FNLT(TSUBST(HD(Y1),Z))=FNLT(TSUBST(HD(X1),Z)) & ¬ISVAR(TSUBST(HD(X1),Z)) &
 ¬ISVAR(TSUBST(HD(Y1),Z)) & SUBST(U,Z)=SUBST(V,Z) &
 ISSUB(Z) & ISTERMLIST(U) & ISTERMLIST(V) & ISTERMLIST(X1) & ISTERMLIST(Y1) &
 ¬Y1=ZERO & ¬X1=ZERO & ISTERM(TSUBST(HD(Y1),Z)) & ISTERM(HD(Y1)) &
 ISTERM(TSUBST(HD(X1),Z)) & ISTERM(HD(X1))
→ APPEND(V,Y1)=APPEND(RCONS(V,HD(Y1)),TL(Y1)) &
  APPEND(U,X1)=APPEND(RCONS(U,HD(X1)),TL(X1)) &
  SUBST(RCONS(U,HD(X1)),Z2*2)=SUBST(RCONS(V,HD(Y1)),Z2*2))

* 10
(¬X1=ZERO & SUBST(U,Z)=SUBST(V,Z) & ISTERMLIST(Y1) & ISTERMLIST(X1) &
 ISTERMLIST(V) & ISTERMLIST(U) & ISTERMLIST(RCONS(U,HD(X1))) &
 ISSUB(Z) & ¬Y1=ZERO & ISTERM(HD(X1)) & ISTERMLIST(RCONS(V,HD(Y1))) &
 ISTERMLIST(TL(Y1)) & ISTERMLIST(TL(X1)) & ISVAR(TSUBST(HD(Y1),Z)) &
 TSUBST(HD(Y1),Z)=TSUBST(HD(X1),Z) & ISTERM(TSUBST(HD(Y1),Z)) & ISTERM(HD(Y1))
→SUBST(RCONS(U,HD(X1)),Z)=SUBST(RCONS(V,HD(Y1)),Z) &
 APPEND(U,X1)=APPEND(RCONS(U,HD(X1)),TL(X1)) &
 APPEND(V,Y1)=APPEND(RCONS(V,HD(Y1)),TL(Y1)))

* 11
(SUBST(U,Z)=SUBST(V,Z) & ISTERMLIST(Y1) & ISTERMLIST(X1) & ISTERMLIST(V) & ISTERMLIST(U) &
 ISSUB(Z) & & ¬X1=ZERO ¬Y1=ZERO & ISVAR(TSUBST(HD(Y1),Z)) & ISTERM(HD(Y1)) &
 ISTERM(TSUBST(HD(X1),Z)) &
 ISTERM(HD(X1)) & ISVAR(TSUBST(HD(X1),Z)) & ISTERM(TSUBST(HD(Y1),Z)) &
 ¬TSUBST(HD(Y1),Z)=TSUBST(HD(X1),Z)
→¬OCCUR(TSUBST(HD(X1),Z),TSUBST(HD(Y1),Z)) &
 (ISTERMLIST(TL(X1)) & ISSUB(COMP(Z,TSUBST(HD(X1),Z),TSUBST(HD(Y1),Z))) &
  ISTERMLIST(RCONS(V,HD(Y1))) & ISTERMLIST(RCONS(U,HD(X1))) & ISTERMLIST(TL(Y1))
 → APPEND(V,Y1)=APPEND(RCONS(V,HD(Y1)),TL(Y1)) &
   APPEND(U,X1)=APPEND(RCONS(U,HD(X1)),TL(X1)) &
   SUBST(RCONS(U,HD(X1)),COMP(Z,TSUBST(HD(X1),Z),TSUBST(HD(Y1),Z)))=SUBST(RCONS(V,HD(Y1)),
   COMP(Z,TSUBST(HD(X1),Z),TSUBST(HD(Y1),Z)))))
```

Figure 7: Some VC's for version 2 in simplified form
The numbering corresponds to the order in which the VC's are generated.

The analysis of the VC's not yet reduced to TRUE shows three areas where the documentation (the basis) has to be extended. Each area is independent from the others, thus they can be dealt with separately. We approach the problem of-proving a VC by first attempting to prove each conjunct in the conclusion separately.

a) OCCUR (VC#11): The conclusion of VC#11 contains ¬OCCUR(A,B). The path of VC#11 is determined by the control tests ISVAR(A), ISVAR(B) and A≠B in its premise. By analysing the path, ¬OCCUR(A,B) is found to be an entry requirement of a call on COMP which was intended under these conditions. So this conjunct of VC#11 is judged correct, and will be satisfied if the user agrees to add the following specification on OCCUR to the basis:

$$\text{IF ISVAR}(X)\wedge\text{ISVAR}(Y)\wedge(Y\neq X)\text{ THEN }\neg\text{OCCUR}(X,Y)=\text{TRUE}$$

b) APPEND (VC's 4,7,9,10,11): As was mentioned before additional properties of APPEND are needed. It turns out that exactly one fact crops up in all the VC's:

$$\text{APPEND}(\text{RCONS}(S,\text{HD}(T)),\text{TL}(T)) = \text{APPEND}(S,T)$$

The programmer might have added a lot of irrelevant properties at 3.1 d) if he had started to write down things about APPEND he thought might be helpful. As seen here, it can be more efficient to write down only very simple axioms and delay anything further until it is seen from the VC's what is needed. If atomic properties of APPEND and RCONS had been added instead, the above fact would have to be deduced from them each time it was required (here: 10 times). It is much more efficient to add the fact to the basis at this point and justify it once during the analysis of the basis (see section 3.4). Moreover, the user can delay completely specifying RCONS.

c) **Equalities involving SUBST** and RCONS (VC's 4,7,9,10,11): As they are the "heart" of the problem the equalities Involving SUBST turn out to be the hardest to get reduced. We could simply assume the properties of SUBST and RCONS that apparently would allow complete reduction to TRUE of all remaining VC's. But, beside the fact that those properties may be too complex to be believable even at the top level, a certain regularity can be observed in the VC's, clue to the structure of the program: The equality in the conclusion is generally of the form

$$(1)\qquad \text{SUBST}(\text{RCONS}(A1,B1),S) = \text{SUBST}(\text{RCONS}(A2,B2),S)$$

whereas the premise includes a corresponding equality

$$(2)\qquad \text{SUBST}(A1,S') = \text{SUBST}(A2,S').$$

Thus, it is sensible to hope that lemmas derived from one problem will be general enough to reduce other problems as well.

Recall that applying a substitution to a list means applying it to each list element separately. So the obvious way to simplify an equality (1) is by reducing it to equality (2) via a statement expressing a kind of commutativity:

$$(3)\ \text{SUBST}(\text{RCONS}(A,B),S) = \text{RCONS}(\text{SUBST}(A,S),\text{TSUBST}(B,S))$$

(the change from SUBST to TSUBST is necessary because of the different type) together with

(4) IF (X1=X2)∧(Y1=Y2) THEN RCONS(X1,Y1)=RCONS(X2,Y2)

as a goal statement. For example, look at the relevant parts of VC*10:

SUBST(U,Z)=SUBST(V,Z) ∧ TSUBST(HD(Y1),Z)=TSUBST(HD(X1),Z)
→ SUBST(RCONS(U,HD(X1)),Z)=SUBST(RCONS(V,HD(Y1)),Z)

Using the statements (3) and (4) the simplifier will generate from the conclusion the subgoal

RCONS(SUBST(U,Z),TSUBST(HD(X1),Z))=RCONS(SUBST(V,Z),TSUBST(HD(Y1),Z))

and from that

SUBST(U,Z)=SUBST(V,Z) ∧ TSUBST(HD(X1),Z)=TSUBST(HD(Y1),Z)

which is just the premise.

Although (3) and (4) will simplify the other VC's further, they are not sufficient to reduce them completely. The equality in VC*4

SUBST(RCONS(U,HD(X1)),Z2*2)=SUBST(RCONS(V,HD(Y1)),Z2*2))

will be reduced to

(5) SUBST(U,Z2*2)=SUBST(V,Z2*2) ∧ TSUBST(HD(X1),Z2*2)=TSUBST(HD(Y1),Z2*2)

Now, the first conjunct obviously has to be proved from the equality

(6) SUBST(U,Z) = SUBST(V,Z)

in the premise. This raises the question, how Z and Z2*2, the actual value of Zl, are related to each other. Looking at the program text we find that Z2 is the substitution returned by the call to UNIFY in case of success; thus, Z2 is an extension of Z by one or more applications of COMP. To express this relationship we introduce the predicate ISSUBSUB(S1:SUB;S2:SUB) meaning "S1 is a sub-substitution of S2" or more precisely: S1 is an initial part of S2 {from which it follows that by composing Sl with appropriate singlesub's we can get S2). We can now formulate a lemma sufficient to reduce the first equality in (5) to (6):

IF ISSUBSUB(Z,Z2) ∧ SUBST(U,Z)=SUBST(V,Z) THEN SUBST(U,Z2)=SUBST(V,Z2)

provided the predicate ISSUBSUB(Z,Z1) is added to the exit condition of UNIFY and therefore also to the invariant of the WHILE loop.

In order to prove the second conjunct of (5) we have to look for "similar" equalities in the premise of VC*4. Obviously, the relevant parts are

(7) SUBST(TERMS(TSUBST(HD(Y1),Z)),Z2*2)=SUBST(TERMS(TSUBST(HD(X1),Z)),Z2*2)
∧ FNLT(TSUBST(HD(Y1),Z))=FNLT(TSUBST(HD(X1),Z))

which exactly mean that TSUBST(HD(X1),Z) and TSUBST(HD(Y1),Z) are unified by Z2*2. If we add as a new axiom (no.22 in figure 8 (appendix)) the condition stating when two functional terms are unified, then (7) will be replaced by: .

(8) TSUBST(TSUBST(HD(Y1),Z),Z2*2)=TSUBST(TSUBST(HD(X1),Z),Z2*2)

The problem now is to prove the second conjunct of (5) from (8). This is a plausible implication and is added as a goal (no.16 in figure 8).

Similarly, other lemmas are derived to reduce the remaining VC's to TRUE.

The third version of the top level program is shown in appendix (figure 8). Using the axioms and goals listed in figure 8 the system is able to reduce all the verification conditions to TRUE except VC*2; this involves more complex propositional structure and is proved easily by the theorem prover. Thus, figure 8 contains an adequate documentation of the top level.

3.4 ANALYSIS OF THE VERIFICATION BASIS. The basis as given in figure 8 is **adequate** to reduce the top level VC's completely, but by no means does the verification of the program end at this point. Beside axioms about data structure primitives the basis contains spcifications on non-primitive functions and lemmas relating these functions.

Analysis of the verification basis is intended to show that the basis is acceptable, that is, we can write programs for the second level functions that satisfy the DEFFUN's and the lemmas. A fairly sensible order of doing this is the following:

1) Axioms from user-defined data structures and standard properties of primitives are accepted,
2) All basis statements involving only primitives must be derived from the standard properties.
3) The number of remaining statements involving second level functions Is reduced by finding dependancies between them.
4) Code for the second level functions is written to satisfy the DEFFUN's and the remaining basis specifications.
5) If a lemma cannot be satisfied, it must be changed. This in turn requires establishing the adequacy of the altered basis for verifying the top level.

Following this scheme, (refering to figure 8 (appendix)) we find that axioms *1 and *2 are part of the data type definitions. (Note that no use was made of other data type axioms so far; however, they will be required to verify lower level functions.) We take the functions APPEND and OCCUR as primitives (standard library functions); axioms nos.3,4,6 are standard properties of them.

Obviously, axiom *1 I follows immediately from axiom *10 and goal *12.

All the remaining basis statements involve second level functions. They obviously cannot be justified using only the given DEFFUN's, but provide further specifications of the subfunctions. They must be regarded as necessary conditions that the programmer's code must satisfy. In this way, they may serve as "guide lines" for the writing of second level programs; some of them - e.g., nos. 9,10,13 - can be translated directly into code as part of the case analysis.

For some of the functions the programs are staightforward. Axioms •5 and •7 specify RCONS: If we define RCONS by

RCONS(X,Y) :≡ APPEND(X,LIST(Y))

then •5 follows easily from well-known properties of APPEND and LIST. Taking COMP as the abbreviation

COMP(S,V,T) :≡ MKSUB(S,PAIR(V,T))

the lemmas nos. 8,9,10, and 12 give the obvious specification of ISSUBSUB in terms of MKSUB.

In appendix figure 10 programs for the second level functions are given which correspond to the DEFFUN's used at the top level. The verification that these programs satisfy the DEFFUN's can be done relative to a basis consisting of the data type definitions (i.e. axioms and DEFFUNS for the constructors and selectors.) This is straightforward since the programs directly reflect the recursive nature of the types.

Remark It should be noted that verification basis for the top level does not necessarily completely predetermine the way second level functions have to be implemented. In our example, application of substitutions can still be either simultaneous or sequential; this solely depends on the representation of the function COMP (or MKSUB). (Although the type definition for SUB implies sequential application, we did not make any use of those axioms.) The implementation in figure 10 assumes sequential application of substitutions.

Now we must show that the programs satisfy the rest of the lemmas, Usually, proving that a lower level function meets a specification (satisfies a lemma in the basis) means setting up a new verification problem by adding the lemma to be justified to the ENTRY and/or EXIT assertions for the body of the function. In complex cases, especially where the proof requires' induction over a data structure, it is necessary to reduce the problem by hand first. (Data structure induction rules are not implemented yet.)

As an example, we show the justification of the goal •15, using the programs from figure 10. First, goal •15 was reduced using the induction rule for the data type SUB to the induction step problem (the base case problem is trivial). This problem in turn was further simplified by hand to (15') using properties of ISSUBSUB and the assumption of the induction step.

(15') ISSUB(S1) ∧ ISSINGLESUB(S2)
⊃ TSUBST(L,MKSUB(S1, S2)) = SINGLETSUBST(TSUBST(L, S1), S2)

If (IS') can be verified, then we can use induction to prove goal 015, Figure 11 (appendix) shows the verification of (15').

Perhaps the reader may be convinced that the proofs of all the remaining lemmas in figure 8 (see Appendix) are as straightforward as •15. Hence figure 8 presents an adequate and acceptable basis (i.e. the lower level functions can indeed be coded to satisfy the lemmas). The top level then is verified.

This is not so.

Goal*16, although simple enough, hides (i.e. depends upon) an extra property of the top level that has not yet come to light. It is not true of substitutions in general, thus it is not acceptable in this form. It is true of the substitutions constructed by the program (which is why it was "believable") because they have a special property. Namely, whenever a variable occurs as the left hand side of a pair, it will not occur in any later pair. This property holds for these substitutions because whenever a substitution for a variable is added to Z, that particular variable is eliminated from all expressions remaining in X I and Y 1 (by then applying Z). The property is equivalent to idempotency of the substitution which we express by the new predicate "IDEM(S)":

$$\mathrm{IDEM}(S) \equiv \mathrm{SUBST}(X,S) = \mathrm{SUBST}(\mathrm{SUBST}(X,S),S) \text{ for all } X.$$

We must change goal *16 to goal *16NEW by adding IDEM(Z) to it as a premiss and then start verification of the top level again (see step 5, beginning Section 3.4). Reasoning along the lines developed in earlier sections (and analysis of the new VC's) shows that we have to 'expand all assertions in the program by appropriate instances of IDEM (see figure 12). Analysis of the VC's shows that one additional lemma is required:

$$\mathrm{IDEM}(S1) \supset \mathrm{IDEM}(\mathrm{COMP}(S1,\mathrm{TSUBST}(X,S1),\mathrm{TSUBST}(Y,S1)))$$

We add this to the basis (goal *16A) and obtain again a complete reduction of the top level VC's.

```
ENTRY ISTERMLIST(X)∧ISTERMLIST(Y)∧ISSUB(Z1)∧IDEM(Z1);
EXIT (ISSUB(Z) ∧ (SUBST(X,Z)=SUBST(Y,Z)) ∧ ISSUBSUB(Z1,Z) ∧ IDEM(Z) ∧ (FLAG=1 ))
      ∨ (FLAG = 0);

INVARIANT (...∧(APPEND(V,Y1)=Y)∧ISSUBSUB(Z1,Z) ∧ IDEM(Z) ∧ (FLAG4 )) ∨ (FLAG=0)

% 16NEW % GOAL TSUBST(@X,@Z)=TSUBST(@Y,@Z)
             SUB ISSUBSUB(@S,Z) ∧ IDEM(@S)
                 ∧ (TSUBST(TSUBST(X,@S),Z)=TSUBST(TSUBST(Y,@S),Z));

%16A% GOAL IDEM(COMP(@S1,TSUBST(@X,@S1),TSUBST(@Y,@S1))) SUB IDEM(S1);
```

Figure 12: Expanded documentation for idempotency

The additional lemma *16A can be justified by showing that it is derivable from standard properties of substitution composition and application. (This proof is given in the appendix.) This means that it will be satisfied by correct code for COMP and TSUBST.

3.5 VERIFICATION OF FURTHER PROPERTIES OF UNIFY. When the user has developed an adequate documentation for his programs with respect to one property, he can attempt to exploit it for the verification of further properties. In this section we demonstrate how additional verification problems can be solved by modifying the established basis and assertions.

The basis developed at the end of section 3.4 (figures 8 and 12) is adequate for verifying a rather weak property of our unification program. However, even this task has brought to light the unusual and useful idempotency property of the substitutions constructed by this program. Now, when we come to verify more stringent requirements we find further code changes to be necessary, and these are justifiabe by idempotency.

Our goal is to verify that
(a) UNIFY generates a most-general-unifier, if the termlists passed as arguments are unifiable;
(b) UNIFY returns FLAG=0, i.e. failure, only if the termlists are not unifiable.

In order to prove (a) we Introduce a predicate

MGU(X,Y,Z) – "S is a most-general-unifier (or mgu, for short) of X and Y, i.e. S is a substitution that unifies X and Y, and if S' is another unifier for X and Y then S is a sub-substitutron of S'."

First of all, assertions in the program are strengthened by replacing all occurrences of equations of the form SUBST(X,S)=SUBST(Y,S) by MGU(X,Y,S). We cannot make a similar simple-minded "strengthening" of the basis since some goal statements are not true if all of the substitutions are restricted to being mgu's. We must find out what properties of MGU need to be added to the existing basis. We therefore return to the verifier and try to derive the necessary axiomatitation for MGU from the VC's.

The first problem arises from a VC corresponding to the path from ENTRY to INVARIANT, which is of the form

$$ISTERMLIST(X) \wedge ISTERMLIST(Y) \wedge ISSUB(Z1) \rightarrow MGU(ZERO, ZERO, Z1) \ldots.$$

This can only be true if Z1=ZERO (see case (iii) in section 2.4). Now, Z1 is a value parameter, So we must ask if Z1 can be eliminated from the body of the procedure.

This leads us to consider the path containing the recursive call to UNIFY, and here we find in general that Z1≠ZERO. The recursive call is of the form

UNIFY(TERMS(X2), TERMS(Y2), Z, Z2, FLAG)

where X2=TSUBST(HD(X1), Z) and Y2=TSUBST(HD(Y1), Z), and the current value of Z replaces the formal parameter Zl. Notice that X2 and Y2 are values resulting from applications of Z to X1 and Y1. If we trace the computation of this call, we find that the only use made of Z (the value of Z1) is to apply it again to X2 and Y2. By idempotency, this second application is redundant! Thus, if we omit the parameter Zl altogether and initialize Z to ZERO, we get exactly the same result by appropriately composing the substitution returned by the recursive call with the old Z. To do this, we introduce a general composition function, SCOMP(Z1,Z2) – the composition of two substitutions, Z1 and Z2.

It turns out that the verification can now be completed by adding one crucial new axiom which describes how to build up mgu's:

(*) MGU(X1,Y1,S1) ∧ MGU(TSUBST(X2,S1), TSUBST(Y2,S1), S2)
⊃ MGU(RCONS(X1,X2), RCONS(Y1,Y2), SCOMP(S1,S2))

(goal #26 in figure 13). If we replace the old COMP(Z,X,Y) by the equivalent MKSUB(Z,PAIR(X,Y)) a special case of (*) (goal #27) will reduce the remaining VC's that involve an updating of Z.

Figure 13 shows the necessary changes to the program and basis for the new problem (compare with figure 8). The remaining new goals are obvious consequences of the definition of MGU. As for acceptability, we give a justification of the not immediately obvious lemma (*) in the appendix.

Remark: Having made these changes (Justified on the basis of idempotency), the reason for having to verify idempotency in the old code disappears, and the required specifications of the new code can be verified without it (see figure 13). Similarly, we no longer need to verify the sub-substitution property.

```
% Goals for MGU %
% 26 % GOAL MGU(RCONS(@X1,@X2),RCONS(@Y1,@Y2),SCOMP(@S1,@S2))
         SUB MGU(X1,Y1,S1)∧MGU(TSUBST(X2,S1),TSUBST(Y2,S1),S2);
% 27 % GOAL MGU(RCONS(@X1,@X2),RCONS(@Y1,@Y2),MKSUB(@S1,@S2))
         SUB MGU(X1,Y1,S1)∧MGU(TSUBST(X2,S1),TSUBST(Y2,S1),S2);
% 28 % GOAL MGU(RCONS(@X1,@X2),RCONS(@Y1,@Y2),@S1)
       · SUB MGU(X1,Y1,S1)∧(TSUBST(X2,S1)=TSUBST(Y2,S1));
% 29A % GOAL MGU(@X,@Y,PAIR(@X,@Y)) SUB ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y);
% 29B % GOAL MGU(@X,@Y,PAIR(@Y,@X)) SUB ISVAR(Y)∧ISTERM(X)∧¬OCCUR(Y,X);
% 30 % GOAL MGU(@X,@Y,@S) SUB (FNLT(X)=FNLT(Y))∧MGU(TERMS(X),TERMS(Y),S);
% 31 % AXIOM MGU(ZERO,ZERO,ZERO)↔TRUE;

DEFFUN MKSUB(S:SUB;S1:SINGLESUB): SUB;
      ENTRY ISSUB(S)∧ISSINGLESUB(S1); EXIT ISSUB(MKSUB);

DEFFUN PAIR(X:VAR; Y:TERM): SINGLESUB;
      ENTRY ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y); EXIT ISSINGLESUB(PAIR);

DEFFUN SCOMP(S1,S2:SUB):SUB;    ENTRY ISSUB(S1)∧ISSUB(S2);     EXIT ISSUB(SCOMP);


PROCEDURE UNIFY (X,Y:TERMLIST; VAR Z:SUB; VAR FLAG:INTEGER);
   ENTRY . . .
   EXIT (ISSUB(Z) ∧ MGU(X,Y,Z) ∧ (FLAG=1 )) ∨ (FLAG=0);

BEGIN
% Initialization of variables %
 . . . Z:=ZERO; . . .

INVARIANT (ISSUB(Z)∧ISTERMLIST(U)∧ISTERMLIST(V)∧ISTERMLIST(X1)
            ∧ ISTERMLIST(Y1)∧MGU(U,V,Z)∧(X=APPEND(U,X1))∧(Y=APPEND(V,Y1))∧ (FLAG=1 ))
          ∨ (FLAG=0)

WHILE . . . DO
  BEGIN
  IF ISVAR(X2) THEN     BEGIN IF ISVAR(Y2)
                              THEN BEGIN IF (X2≠Y2) THEN Z:=MKSUB(Z,PAIR(X2,Y2))
                                         END
                               ELSE . . . .
                        END
  ELSE   BEGIN IF ISVAR(Y2) THEN . . . .
                  ELSE BEGIN    IF FNLT(X2)=FNLT(Y2)
                                THEN  BEGIN UNIFY(TERMS(X2),TERMS(Y2),Z2,FLAG);
                                            IF FLAG=1 THEN Z:=SCOMP(Z,Z2)
                                            ELSE FLAG:=0
                                      END
                                ELSE . . . .
```

Figure 13: Additional documentation and program changes for MCU

Problem (b) is to show that if UNIFY returns FLAG-O then there is no unifier for X and Y. We express this property by the predicate:

NOTUNIF(X,Y) – "X and Y are not unifiable"

and set up the new verification problem

$$\ldots \{\mathrm{UNIFY(X,Y,Z,FLAG)}\}(\ldots \wedge \mathrm{FLAG}=1) \vee (\mathrm{FLAG}=0 \wedge \mathrm{NOTUNIF(X,Y)})$$

Note that the post-condition implies "FLAG=1<=>NOTUNIF(X,Y)." The adequate axiomatization of NOTUNIF is almost straightforward. However, in goals *33 and *34 (see figure 14) the premiss MGU(. . .) is crucial for acceptability.

The final program and documentation for the full verification of UNIFY is given in figure 14.

ACKNOWLEDGEMENTS. We wish to acknowledge the efforts of M. Sintzoff in helping to set up this experiment at the beginning. N. Suzuki has cooperated at all stages in modifying the simplification system to handle our problems.

REFERENCES

[Boyer and Moore] R.S. Boyer and J S. Moore, "Proving Theorems about LISP Problems", Third IJCAI Proceedings, 1973.

[Clint] M.Clint, "Program Proving: Coroutines", Acta Informatica 2(1973), 52-63.

[Deutsch] L.P. Deutsch, "An Interactive Program Verifier", Ph.D. thesis, University of California, Berkeley, 1973.

[ELW] B. Elspas, K. Levitt, and R. Waldinger, "An Interactive System for the Verification of Computer Programs", SRI Project 1891, Stanford Research Institute, 1973.

[Floyd] R. W. Floyd, "Algorithms 245. TREESORT3", Comm.ACM 7 (1964).

[Good and Ragland] D. Good and L. Ragland, "NUCLEUS - A Language of Provable Programs", in Program Test Methods, W.Hetzel (ed.), Prentice Hall, 1973.

[Hoare71] C.A.R. Hoare, "Procedures and parameters: An axiomatic approach", in Symposium on Semantics of Algorithmic Languages, Engeler, E. (ed.), Springer-Verlag, 1971, pp. 102-116.

[Hoare1973] C.A.R. Hoare, "Recursive data types", Memo AIM-223, Stanford Artificial Intelligence Project, Stanford University, 1973.

[Hoare and Wirth] C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCA L", Acta Informatica 2 (1973), 335-355.

[ILL] S. Igarashi, R.L. Loncion, and D.C. Luckham, "Automatic Program Verification I: Logical Basis and Its Implementation", AIM-200, Stanford Artificial Intelligence Project, Stanford University, 1972.

[King and Floyd] J.C. King and R.W. Floyd, " An Interpretation Oriented Theorem Prover over the Integers", Journal of Comp. and Sys.Sci., vol.6, no.4, Aug. 1972 pp.305-323.

[McCarthy] J. McCarthy, "A basis for a mathematical theory of computation", in Computer Programming and Formal Systems, (ed. Braffort and Hirschberg), North Holland, 1963.

. [Morales] J. Morales, "Verification of Sorting A lgorithms", unpublished report, Sept 1973.

[Suzuki] N. Suzuki, "Verification of Programs by Algebraic and Logical Reduction", forthcoming Memo, Stanford Artificial Intelligence Project, Stanford University.

[Wirth] N. Wirth, "The programming language Pascal", Acta Informatica 1.1 (197 l), 35-63.

APPENDIX

Proofs of Lemmas

Notation: We use the following short-hand notation:

"$x.\alpha$"	for substitution application (TSUBST/SUBST)
"$\alpha \otimes \beta$"	for substitution composition (MKSUB/SCOMP)
"$x \mathbin{::} y$"	for list concatenation (APP/RCONS)
"$<x,y>$"	for PAIR(X,Y)

We make use of certain facts about substitutions:

(i) associativity of "$\otimes$": $\alpha 1 \otimes (\alpha 2 \otimes \alpha 3) = (\alpha 1 \otimes \alpha 2) \otimes \alpha 3$
(ii) a kind of associativity of ".": $(x.\alpha 1).\alpha 2 = x.(\alpha 1 \otimes \alpha 2)$
(iii) a kind of distributivity of ".": $(x \mathbin{::} y).\alpha = (x.\alpha) \mathbin{::} (y.\alpha)$

1. Coal 16A: $\mathrm{IDEM}(\alpha) \rightarrow \mathrm{IDEM}(\alpha \otimes <x.\alpha, y.\alpha>)$

This is equivalent to proving

$$(\alpha \otimes <x.\alpha, y.\alpha>) \otimes (\alpha \otimes <x.\alpha, y.\alpha>) = \alpha \otimes <x.\alpha, y.\alpha>$$

from the assumptions

(a1) $\alpha \otimes \alpha = \alpha$, (a2) isvar($x.\alpha$), (a3) isterm($y.\alpha$), (a4) $\neg$ occur($x.\alpha$,$y.\alpha$)

(a2)-(a4) are from the ENTRY assertion for COMP; they imply that the single substitution is idempotent, namely

(b) $<x.\alpha, y.\alpha> \otimes <x.\alpha, y.\alpha> = <x.\alpha, y.\alpha>$

Now
$(\alpha \otimes <x.\alpha, y.\alpha>) \otimes (\alpha \otimes <x.\alpha, y.\alpha>)$
$= ((\alpha \otimes <x.\alpha, y.\alpha>) \otimes \alpha) \otimes <x.\alpha, y.\alpha>$ by (i)
$= [(\alpha \otimes \alpha) \otimes <x.\alpha, y.(\alpha \otimes \alpha>)] \otimes <x.\alpha, y.\alpha>$ using standard properties of "." and "$\otimes$" and (al)
$= [\alpha \otimes <x.\alpha, y.\alpha>] \otimes <x.\alpha, y.\alpha>$ by (a)
$= \alpha \otimes <x.\alpha, y.\alpha>$ by (b), (i)

2. LEMMA $(\ast)$ for MGU (Section 3.5)

$(\ast)$ $\mathrm{MGU}(X1, Y1, \alpha 1) \wedge \mathrm{MGU}(X2.\alpha 1, Y2.\alpha 1, \alpha 2) \supset \mathrm{MGU}(X1 \ast X2, Y1 \ast Y2, \alpha 1 \circledast \alpha 2)$

We prove $(\ast)$ from the assumptions

(1)	$\mathrm{MGU}(X1, Y1, \alpha 1)$	
and		
(2)	$\mathrm{MGU}(X2.\alpha 1, Y2.\alpha 1, \alpha 2)$	
Suppose	$(X1 \ast X2).\beta = (Y1 \ast Y2).\beta$	
Then by (1)	$\beta = \alpha 1 \circledast \beta 1$	for suitable $\beta 1$,
so	$(X1 \ast X2).(\alpha 1 \circledast \beta 1) = (Y1 \ast Y2).(\alpha 1 \circledast \beta 1)$	
which implies	$[(X1.\alpha 1) \ast (X2.\alpha 1)].\beta 1 = [(Y1.\alpha 1) \ast (Y2.\alpha 1)].\beta 1$	by (ii), (iii)
From this we infer	$(X2.\alpha 1).\beta 1 = (Y2.\alpha 1).\beta 1$	
Thus, by (2)	$\beta 1 = \alpha 2 \circledast \beta 2$	
or	$\beta = (\alpha 1 \circledast \alpha 2) \circledast \beta 2$	

for suitable $\beta 2$, which proves $(\ast)$.

GOALFILE

```
% Axioms defining the data types and basic functions %

% 1 % AXIOM ISTERMLIST(ZERO)↔ TRUE;
% 2 % AXIOM ISSUB(ZERO)↔TRUE;

% Axioms describing properties of subfunctions %

% 3 % AXIOM APPEND(ZERO,@S)↔S;
% 4 % AXIOM APPEND(@S,ZERO)↔S;

%  5  % AXIOM APPEND(RCONS(@S,HD(@T)),TL(@T))↔APPEND(S,T);

% 6 % AXIOM IF ISVAR(X)∧ISVAR(Y)∧(Y≠X) THEN ¬OCCUR(@X,@Y)↔TRUE;

%  7 % GOAL RCONS(@X1,@X2)=RCONS(@Y1,@Y2) SUB (X1 =Y1)∧(X2=Y2);

% 8 % AXIOM IF ISSUB(Z) THEN ISSUBSUB(ZERO,@Z)↔TRUE;

% 9 % AXIOM ISSUBSUB(@Z,@Z)↔TRUE;

%10 % AXIOM ISSUBSUB(@Z,COMP(@Z,@X,@Y))↔TRUE;

% 11% AXIOM IF ISSUBSUB(Y,Z) THEN ISSUBSUB(@Y,COMP(@Z,@V,@W))↔TRUE;

%12% GOAL ISSUBSUB(@Z,@Z1) SUB ISSUBSUB(@Z2,Z1)∧ISSUBSUB(Z,@Z2),
                              ISSUBSUB(Z,@Z2)∧ISSUBSUB(@Z2,Z1);

% 13 % AXIOM TSUBST(@X,ZERO)↔X;

% 14 % AXIOM IF ¬ISVAR(X) THEN ¬ISVAR(TSUBST(@X,@S))↔TRUE;

%15% GOAL TSUBST(@X,@Z)=TSUBST(@Y,@Z) SUB ISSUBSUB(@Z1,Z)∧(TSUBST(X,@Z1)=TSUBST(Y,@Z1));

% 16% GOAL TSUBST(@X,@Z)=TSUBST(@Y,@Z)
        SUB ISSUBSUB(@S,Z)∧(TSUBST(TSUBST(X,@S),Z)=TSUBST(TSUBST(Y,@S),Z));

% 17 % GOAL TSUBST(@X,COMP(@Z,@X,@Y))=TSUBST(@Y,COMP(@Z,@X,@Y))
        SUB ISSUB(COMP(Z,X,Y)), ISSUB(COMP(Z,Y,X));

%18 % GOAL TSUBST(@X,COMP(@Z,@U,@V))=TSUBST(@Y,COMP(@Z,@U,@V))
        SUB (U=TSUBST(X,Z))∧(V=TSUBST(Y,Z))∧ISSUB(COMP(Z,U,V)),
            (U=TSUBST(Y,Z))∧(V=TSUBST(X,Z))∧ISSUB(COMP(Z,U,V));
```

Figure 8 (continued)

```
% 19 % AXIOM SUBST(@X,ZERO)↔X;

% 20 % AXIOM SUBST(ZERO,@S)↔ZERO;

% 21 % AXIOM SUBST(RCONS(@X,@Y),@Z)↔RCONS(SUBST(X,Z),TSUBST(Y,Z));

% 22 % AXIOM IF FNLT(X)=FNLT(Y) THEN (SUBST(TERMS(@X),@Z)=SUBST(TERMS(@Y),@Z))↔
                                (TSUBST(X,Z)=TSUBST(Y,Z));

% 23 % GOAL SUBST(@X,COMP(@Z,@A,@B))=SUBST(@Y,COMP(@Z,@A,@B))
         SUB (SUBST(X,Z)=SUBST(Y,Z))∧ISSUB(COMP(Z,A,B));

% 24 % GOAL SUBST(@X,@Z)=SUBST(@Y,@Z)  SUB ISSUBSUB(@Z1,Z)∧(SUBST(X,@Z1)=SUBST(Y,@Z1));.;


PASCAL

DEFFUN HD(L:TERMLIST):TERM;   ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERM(HD);

DEFFUN TL(L:TERMLIST):TERMLIST; ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERMLIST(TL);

DEFFUN RCONS(L:TERMLIST;X:TERM):TERMLIST;
         ENTRY ISTERMLIST(L)∧ISTERM(X); EXIT ISTERMLIST(RCONS);

DEFFUN TERMS(X:TERM):TERMLIST; ENTRY ISTERM(X)∧¬ISVAR(X);   EXIT ISTERMLIST(TERMS);

DEFFUN FNLT(X:TERM):CONST;   ENTRY ISTERM(X)∧¬ISVAR(X);   EXIT ISCONST(FNLT);

DEFFUN TSUBST(X:TERM;S:SUB):TERM; ENTRY ISTERM(X)∧ISSUB(S);   EXIT ISTERM(TSUBST);

DEFFUN SUBST(X:TERMLIST; S:SUB):TERMLIST;
         ENTRY ISTERMLIST(X)∧ISSUB(S); EXIT ISTERMLIST(SUBST);

DEFFUN COMP(S:SUB; X:VAR; Y:TERM):SUB;
         ENTRY ISSUB(S)∧ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y);   EXIT ISSUB(COMP);

DEFFUN OCCUR(X:VAR; Y:TERM):BOOLEAN; ENTRY ISVAR(X)∧ISTERM(Y); EXIT ISBOOLEAN(OCCUR);
```

Figure 8 **(continued)**

```
PROCEDURE UNIFY (X,Y:TERMLIST; Z1:SUB; VAR Z:SUB; VAR FLAG:BOOLEAN);
   ENTRY ISTERMLIST(X)∧ISTERMLIST(Y)∧ISSUB(Z1);
   EXIT (ISSUB(Z)∧(SUBST(X,Z)=SUBST(Y,Z))∧ISSUBSUB(Z1,Z)∧(FLAG=1)) v (FLAG = 0);

VAR U,V,X1,Y1:TERMLIST; VAR X2,Y2:TERM; VAR Z2:SUB;
BEGIN

% Initialization of variables %
U:=ZERO; V:=ZERO; Z:=Z1; X1:=X; Y1:=Y; FLAG:=1;

INVARIANT (ISSUB(Z)∧ISTERMLIST(U)∧ISTERMLIST(V)∧ISTERMLIST(X1)∧ISTERMLIST(Y1)∧
       (SUBST(U,Z)=SUBST(V,Z))∧(APPEND(U,X1)=X)∧(APPEND(V,Y1)=Y)∧ISSUBSUB(Z1,Z)∧(FLAG=1))
     v (FLAG=0)

WHILE (X1≠ZERO) ∧(Y1≠ZERO)∧ (FLAG=1) DO
  BEGIN
   X2:=TSUBST(HD(X1),Z);
   Y2:=TSUBST(HD(Y1),Z);
   IF ISVAR(X2) THEN    BEGIN IF ISVAR(Y2)
                              THEN BEGIN IF (X2≠Y2)
                                            THEN Z:=COMP(Z,X2,Y2)
                                     END
                              ELSE BEGIN IF OCCUR(X2,Y2) THEN FLAG:=0
                                            ELSE Z:=COMP(Z,X2,Y2)
                                     END
                        END
    ELSE BEGIN IF ISVAR(Y2)
                THEN BEGIN IF OCCUR(Y2,X2) THEN FLAG :=0
                              ELSE Z:=COMP(Z,Y2,X2)
                     END
                ELSE BEGIN IF FNLT(X2)=FNLT(Y2)
                              THEN BEGIN UNIFY(TERMS(X2),TERMS(Y2),Z,Z2,FLAG);
                                            IF FLAG=1 THEN Z:=Z2
                                     END
                              ELSE FLAG:=0
                     END
         END;
   U:=RCONS(U,HD(X1)); V:=RCONS(V,HD(Y1));
   X1:=TL(X1);          Y1:=TL(Y1);
  END; % End of WHILE body %

IF (X1≠ZERO) v (Y1≠ZERO) THEN FLAG:=0
END; % Procedure body %
```

Figure 8: Third version with documentation

FOR **UNIFY** THERE ARE 11 VERIFICATION CONDITIONS. HERE IS ONE OF THEM:

```
* 4
(¬X1=ZERO & ¬Y1=ZERO & FLAG=1 &
 ISSUB(Z)∧ISTERMLIST(U)∧ISTERMLIST(V)∧ISTERMLIST(X1)∧ISTERMLIST(Y1)∧SUBS~
 T(U,Z)=SUBST(V,Z)∧APPEND(U,X1)=X∧APPEND(V,Y1)=Y∧ISSUBSUB(Z1,Z)∧FLAG=1∨FLAG=0
→ISTERMLIST(X1) & ¬X1=ZERO & (ISTERM(HD(X1))
 → ISTERM(HD(X1)) & ISSUB(Z) & (ISTERM(TSUBST(HD(X1),Z))
  →ISTERMLIST(Y1) & ¬Y1=ZERO & (ISTERM(HD(Y1))
   → ISTERM(HD(Y1)) & ISSUB(Z) & (ISTERM(TSUBST(HD(Y1),Z))
    → (¬ISVAR(TSUBST(HD(Y1),Z)) & ¬ISVAR(TSUBST(HD(X1),Z))
     → ISTERM(TSUBST(HD(Y1),Z)) & ¬ISVAR(TSUBST(HD(Y1),Z)) &
      (ISCONST(FNLT(TSUBST(HD(Y1),Z)))
      → ISTERM(TSUBST(HD(X1),Z)) & ¬ISVAR(TSUBST(HD(X1),Z)) &
       (ISCONST(FNLT(TSUBST(HD(X1),Z)))
       → (FNLT(TSUBST(HD(X1),Z))=FNLT(TSUBST(HD(Y1),Z))
        → ISTERM(TSUBST(HD(Y1),Z)) & ¬ISVAR(TSUBST(HD(Y1),Z)) &
         (ISTERMLIST(TERMS(TSUBST(HD(Y1),Z)))
         → ISTERM(TSUBST(HD(X1),Z)) & ¬ISVAR(TSUBST(HD(X1),Z)) &
          (ISTERMLIST(TERMS(TSUBST(HD(X1),Z)))
          →ISTERMLIST(TERMS(TSUBST(HD(X1),Z)))&ISTERMLIST(TERMS(TSUBST(HD(Y1),Z))) &
           ISSUB(Z) & (FLAG=1 &
           ISSUB(Z2*2)∧SUBST(TERMS(TSUBST(HD(X1),Z)),Z2*2)=SUBST(TERMS(~
           TSUBST(HD(Y1),Z)),Z2*2)∧ISSUBSUB(Z,Z2*2)∧FLAG=1∨FLAG=0
           →ISTERMLIST(X1) & ¬X1=ZERO & (ISTERM(HD(X1))
            → ISTERMLIST(U) & ISTERM(HD(X1)) & (ISTERMLIST(RCONS(U,HD(X1)))
             → ISTERMLIST(Y1) & ¬Y1=ZERO & (ISTERM(HD(Y1))
              → ISTERMLIST(V) & ISTERM(HD(Y1)) & (ISTERMLIST(RCONS(V,HD(Y1)))
               → ISTERMLIST(X1) & ¬X1=ZERO & (ISTERMLIST(TL(X1))
                → ISTERMLIST(Y1) & ¬Y1=ZERO & (ISTERMLIST(TL(Y1))
                 →ISSUB(Z2*2)∧ISTERMLIST(RCONS(U,HD(X1)))∧ISTERMLIST(RCONS(V,HD(Y1)))∧
                  ISTERMLIST(TL(X1))∧ISTERMLIST(TL(Y1))∧
                  SUBST(RCONS(U,HD(X1)),Z2*2)=SUBST(RCONS(V,HD(Y1)),Z2*2)∧
                  APPEND(RCONS(U,HD(X1)),TL(X1))=X∧
                  APPEND(RCONS(V,HD(Y1)),TL(Y1))=Y∧
                  ISSUBSUB(Z1,Z2*2)∧FLAG=1∨FLAG=0)))))))))))))))))
```

Figure 9: One of the unsimplified VC's for the third version

```
GOALFILE

AXIOM ISTERMLIST(ZERO)↔ TRUE;

AXIOM ISSUB(ZERO)↔ TRUE;

AXIOM ISSINGLESUB(ZERO)↔ TRUE;

GOAL ISSINGLESUB(@S) SUB (S=PAIR(@X,@Y))∧ISVAR(@X)∧ISTERM(@Y)∧¬OCCUR(@X,@Y);.;

PASCAL

DEFFUN MKTERM(X:CONST;Y:TERMLIST):TERM;
        ENTRY ISCONST(X)∧ISTERMLIST(Y);       EXIT ISTERM(MKTERM);

DEFFUN FNLT(X:TERM):CONST;
        ENTRY ISTERM(X)∧¬ISVAR(X);    EXIT ISCONST(FNLT);

DEFFUN TERMS(X:TERM):TERMLIST;
        ENTRY ISTERM(X)∧¬ISVAR(X);    EXIT ISTERMLIST(TERMS);

DEFFUN CONS(X:TERM; L:TERMLIST):TERMLIST;
        ENTRY ISTERM(X)∧ISTERMLIST(L); EXIT ISTERMLIST(CONS);

DEFFUN HD(L:TERMLIST):TERM;
        ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERM(HD);

DEFFUN TL(L:TERMLIST):TERMLIST;
        ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERMLIST(TL);

DEFFUN MKSUB(S:SUB; S1:SINGLESUB):SUB;
        ENTRY ISSUB(S)∧ISSINGLESUB(S1); EXIT ISSUB(MKSUB);

DEFFUN LAST(S:SUB):SINGLESUB;
        ENTRY ISSUB(S);                EXIT ISSINGLESUB(LAST);

DEFFUN REST(S:SUB):SUB;
        ENTRY ISSUB(S);                EXIT ISSUB(REST);

DEFFUN VAR(S:SINGLESUB):VAR;
        ENTRY ISSINGLESUB(S);          EXIT ISVAR(VAR);

DEFFUN TERM(S:SINGLESUB):TERM;
        ENTRY ISSINGLESUB(S); EXIT ISTERM(TERM);

DEFFUN PAIR(X:VAR; Y:TERM):PAIR;
        ENTRY ISVAR(X)∧ISTERM(Y);       EXIT ISPAIR(PAIR);
```

Figure 10 (continued)

```
FUNCTION SUBST(L:TERMLIST;S:SUB):TERMLIST;
   ENTRY ISTERMLIST(L) ∧ ISSUB(S);
   EXIT ISTERMLIST(SUBST);

BEGIN
IF (S=ZERO) THEN SUBST:=L
 ELSE SUBST:=SINGLESUBST(SUBST(L,REST(S)),LAST(S));
END;


FUNCTION TSUBST(X:TERM;S:SUB):TERM;
   ENTRY ISTERM(X) ∧ ISSUB(S);
   EXIT ISTERM(TSUBST);

BEGIN
IF (S=ZERO) THEN TSUBST:=X
 ELSE TSUBST:=SINGLETSUBST(TSUBST(X,REST(S)),LAST(S));
END;


FUNCTION SINGLESUBST(L:TERMLIST;S:SINGLESUB):TERMLIST;
   ENTRY ISTERMLIST(L) ∧ ISSINGLESUB(S);
   EXIT ISTERMLIST(SINGLESUBST);

BEGIN
IF (L=ZERO) THEN SINGLESUBST:=ZERO
 ELSE SINGLESUBST:=CONS(SINGLETSUBST(HD(L),S),SINGLESUBST(TL(L),S))
END;


FUNCTION SINGLETSUBST(T:TERM; S:SINGLESUB):TERM;
   ENTRY ISTERM(T) ∧ ISSINGLESUB(S);
   EXIT ISTERM(SINGLETSUBST);

BEGIN
IF ISVAR(T) THEN BEGIN IF (T=VAR(S))
                      THEN SINGLETSUBST := TERM(S)
                      ELSE SINGLETSUBST := T
                 END
ELSE SINGLETSUBST :=MKTERM(FNLT(T),SINGLESUBST(TERMS(T), S))
END;


FUNCTION COMP(S:SUB; X:VAR; Y:TERM):SUB;
  ENTRY ISSUB(S)∧ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y);
  EXIT ISSUB(COMP);

BEGIN COMP:=MKSUB(S,PAIR(X,Y)); END;
```

Figure JO: second level functions atrd goalfile

```
% Program for verifying
        ISSUB(S1)∧ISSINGLESUB(S2)∧ISTERMLIST(L)
        ⊃[TSUBST(L,MKSUB(S1,S2))=SINGLETSUBST(TSUBST(L,S1),S2)]
Program body of TSUBST with new entry/exit conditions %

PASCAL
ENTRY ISSUB(S)∧(S=MKSUB(S1,S2))∧ISTERM(X);
EXIT (TSUBST=SINGLETSUBST(TSUBST(X,S1),S2));

AXIOM LAST(MKSUB(@S,@S1))↔S1;
AXIOM REST(MKSUB(@S,@S1))↔S;
AXIOM (ZERO=MKSUB(@S1,@S2))↔FALSE;

BEGIN
IF (S=ZERO) THEN TSUBST:=X ELSE TSUBST:=SINGLETSUBST(TSUBST(X,REST(S)),LAST(S));
END.;


FOR THE MAIN PROGRAM THERE ARE 2 VERIFICATION CONDITIONS

# 1
(S=ZERO & ISSUB(S) & S=MKSUB(S1,S2) & ISTERM(X)
→X=SINGLETSUBST(TSUBST(X,S1),S2))

# 2
(¬S=ZERO & ISSUB(S) & S=MKSUB(S1,S2) & ISTERM(X)
→ SINGLETSUBST(TSUBST(X,REST(S)),LAST(S))=SINGLETSUBST(TSUBST(X,S1),S2))

AFTER SOME SIMPLIFICATION, YOU CAN GET

# 1 TRUE
# 2 TRUE
```

Figure 11: lemma about goal 15

GOALFILE

```
% Axioms defining the data types and basic functions %
%1% AXIOM ISTERMLIST(ZERO)↔TRUE;
% 2 % AXIOM ISSUB(ZERO)↔TRUE;

% Axioms describing properties of subfunctions %
% 3 % AXIOM APPEND(ZERO,@S)↔S;
% 4 % AXIOM APPEND(@S,ZERO)↔S;
% 5 % AXIOM APPEND(RCONS(@S,HD(@T)),TL(@T))↔APPEND(S,T);
% 6 % AXIOM IF ISVAR(X)∧ISVAR(Y)∧(Y≠X) THEN ¬OCCUR(@X,@Y)↔TRUE;
% 13 % AXIOM TSUBST(@X,ZERO)↔X;
% 14 % AXIOM IF ¬ISVAR(X) THEN ¬ISVAR(TSUBST(@X,@S))↔TRUE;

% Goals for MGU %

% 26 % GOAL MGU(RCONS(@X1,@X2),RCONS(@Y1,@Y2), SCOMP(@S1,@S2))
      SUB MGU(X1,Y1,S1)∧MGU(TSUBST(X2,S1),TSUBST(Y2,S1),S2);

% 27 % GOAL MGU(RCONS(@X1,@X2),RCONS(@Y1,@Y2), MKSUB(@S1,@S2))
      SUB MGU(X1,Y1,S1)∧MGU(TSUBST(X2,S1),TSUBST(Y2,S1),S2);

% 28 % GOAL MGU(RCONS(@X1,@X2),RCONS(@Y1,@Y2),@S 1)
      SUB         MGU(X   )∧(TSUBST(X2,S1)=TSUBST(Y2,S 1));

% 29A % GOAL MGU(@X,@Y,PAIR(@X,@Y)) SUB ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y);
% 29B % GOAL MGU(@X,@Y,PAIR(@Y,@X)) SUB ISVAR(Y)∧ISTERM(X)∧¬OCCUR(Y,X);

% 30 % GOAL MGU(@X,@Y,@S) SUB (FNLT(X)=FNLT(Y))∧MGU(TERMS(X),TERMS(Y),S);

% 31% AXIOM MGU(ZERO,ZERO,ZERO)↔TRUE;

% Goals for NOTUNIF %

% 32 % GOAL NOTUNIF(@X,@Y)
      SUB ISVAR(X)∧ISTERM(Y)∧¬ISVAR(Y)∧OCCUR(X,Y),
          ISTERM(X)∧ISTERM(Y)∧¬ISVAR(X)∧¬ISVAR(Y)∧ ¬(FNLT(X)=FNLT(Y)),
          (FNLT(X)=FNLT(Y))∧NOTUNIF(TERMS(X),TERMS(Y));

% 33 % GOAL NOTUNIF(RCONS(@X1,@X2),RCONS(@Y1,@Y2))
      SUB MGU(X1,Y1,@S)∧NOTUNIF(TSUBST(X2,@S),TSUBST(Y2,@S)),
          MGU(X1,Y1,@S) ∧ NOTUNIF(TSUBST(Y2,@S),TSUBST(X2,@S)),
          NOTUNIF(X2,Y2);

% 34 % GOAL NOTUNIF(APPEND(@X1,@X2),APPEND(@Y1,@Y2))
      SUB (X2=ZERO)∧¬(Y2=ZERO)∧MGU(X1,Y1,@S),
          (Y2=ZERO)∧¬(X2=ZERO)∧MGU(X1,Y1,@S),
          NOTUNIF(X1,Y1);
```

Figure 14 (continued)

```
PASCAL

DEFFUN HD(L:TERMLIST):TERM;     ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERM(HD);

DEFFUN TL(L:TERMLIST):TERMLIST; ENTRY ISTERMLIST(L)∧¬(L=ZERO); EXIT ISTERMLIST(TL);

DEFFUN RCONS(L:TERMLIST;X:TERM):TERMLIST;
       ENTRY ISTERMLIST(L)∧ISTERM(X); EXIT ISTERMLIST(RCONS);

DEFFUN TERMS(X:TERM):TERMLIST; ENTRY ISTERM(X)∧¬ISVAR(X);    EXIT ISTERMLIST(TERMS);

DEFFUN FNLT(X:TERM):CONST;     ENTRY ISTERM(X)∧¬ISVAR(X);    EXIT  ISCONST(FNLT);

DEFFUN TSUBST(X:TERM;S:SUB):TERM; ENTRY ISTERM(X)∧ISSUB(S); EXIT ISTERM(TSUBST);

DEFFUN MKSUB(S:SUB; S1:SINGLESUB):SUB;
       ENTRY ISSUB(S)∧ISSINGLESUB(S1); EXIT ISSUB(MKSUB);

DEFFUN PAIR(X:VAR; Y:TERM):SINGLESUB;
       ENTRY ISVAR(X)∧ISTERM(Y)∧¬OCCUR(X,Y); EXIT ISSINGLESUB(PAIR);

DEFFUN SCOMP(S1,S2:SUB):SUB;   ENTRY ISSUB(S1)∧ISSUB(S2);    EXIT ISSUB(SCOMP);

DEFFUN OCCUR(X:VAR; Y:TERM):BOOLEAN;   ENTRY ISVAR(X)∧ISTERM(Y); EXIT ISBOOLEAN(OCCUR);
```

Figure 14 (continued)

```
PROCEDURE UNIFY (X,Y:TERMLIST; VAR Z:SUB; VAR FLAG:INTEGER);
   ENTRY ISTERMLIST(X)∧ISTERMLIST(Y);
   EXIT (ISSUE(Z) ∧ MGU(X,Y,Z) ∧ (FLAG=1)) ∨ ((FLAG=0) ∧ NOTUNIF(X,Y));

BEGIN

% Initialization of variables %
U:=ZERO; V:=ZERO; Z:=ZERO;    X1:=X;    Y1:=Y;   FLAG:=1;

INVARIANT (ISSUB(Z)∧ISTERMLIST(U)∧ISTERMLIST(V)∧ISTERMLIST(X1)
        ∧ISTERMLIST(Y1)∧MGU(U,V,Z)∧(X=APPEND(U,X1))∧(Y=APPEND(V,Y1))∧(FLAG=1))
      ∨ ((FLAG=0)∧NOTUNIF(U,V)∧(X=APPEND(U,X1))∧(Y=APPEND(V,Y1)))

WHILE (X1≠ZERO) ∧(Y1≠ZERO) ∧(FLAG=1) DO
  BEGIN
   X2:= TSUBST(HD(X1),Z);
   Y2:= TSUBST(HD(Y1),Z);
   IF ISVAR(X2) THEN    BEGIN IF ISVAR(Y2)
                              THEN BEGIN IF (X2≠Y2)
                                         THEN Z:=MKSUB(Z,PAIR(X2,Y2))
                                   END
                              ELSE BEGIN IF OCCUR(X2,Y2) THEN FLAG:=0
                                         ELSE Z:=MKSUB(Z,PAIR(X2,Y2))
                                   END
                        END
   ELSE BEGIN IF ISVAR(Y2)
              THEN BEGIN IF OCCUR(Y2,X2) THEN FLAG:=0
                              ELSE Z:=MKSUB(Z,PAIR(Y2,X2))
                        END
              ELSE BEGIN IF FNLT(X2)=FNLT(Y2)
                              THEN BEGIN UNIFY(TERMS(X2),TERMS(Y2),Z2,FLAG);
                                         IF FLAG=1 THEN Z:=SCOMP(Z,Z2)
                                          ELSE FLAG:=0
                                   END
                              ELSE FLAG:=0
                        END
        END;
   U :=RCONS(U,HD(X1));
   V :=RCONS(V,HD(Y1));
   X1 :=TL(X1);
   Y1 :=TL(Y1);
  END; % End of WHILE body %

IF (X1≠ZERO) ∨ (Y1≠ZERO) THEN FLAG:=0
END; % Procedure body %

THE VERIFICATION TAKES 286 CPU SEC.
```

Figure 14: The complete program and documentation for UNIFY