THE UNIVERSITY *of* EDINBURGH

Interactive program verification using virtual programs

by

Rodney W. Topor

Ph.D.

University of Edinburgh

1975

ABSTRACT

This thesis is concerned with ways of proving the correctness of computer programs. The first part of the thesis presents a new method for doing this. The method, called continuation induction, is based on the ideas of symbolic execution, the description of a given program by a virtual program, and the demonstration that these two programs are equivalent whenever the given program terminates. The main advantage of continuation induction over other methods is that it enables programs using a wide variety of programming constructs such as recursion, iteration, non-determinism, procedures with side-effects and jumps out of blocks to be handled in a natural and uniform way.

In the second part of the thesis a program verifier which uses both this method and Floyd's inductive assertion method is described. The significance of this verifier is that it is designed to be extensible, and to this end the user can declare new functions and predicates to be used in giving a natural description of the program's intention. Rules describing these new functions can then be used when verifying the program. To actually prove the verification conditions, the system employs automatic simplification, a relatively clever matcher, a simple natural deduction system and, most importantly, the user's advice. A large number of commands are provided for the user in guiding the system to

a proof of the program's correctness. The system has been used to verify various programs including two sorting programs and a program to invert a permutation "in place"; the proofs of the sorting programs included a proof of the fact that the final array was a permutation of the original one. Finally, some observations and suggestions are made concerning the continued development of such interactive verification systems.

ACKNOWLEDGEMENTS

I would like to express my appreciation to the following people and organizations for helping me in their different ways:

- to my supervisor, Rod Burstall, for advice and encouragement, and for introducing me to the subject of program verification,

- to Gordon Plotkin and Jerry Schwarz for their critical comments on a draft of the thesis,

- to Raymond Aubin, John Darlington and Mike Gordon for useful discussions on the subjects of program verification and manipulation,

- to Eleanor Kerse for helping with the typing and for struggling so valiantly to help with the editing of this thesis,

- to Harry Barrow, Robin Popplestone and all the other members of the Department of Artificial Intelligence for creating such a stimulating, if excessively political, atmosphere in which to work,

- to the Commonwealth Scholarship Commission and, in the later stages, the Science Research Council for financial support.

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

1.1 Overview

This thesis describes a method and an implemented system for proving the correctness of computer programs. The task of proving programs correct, known as *program verification*, is one aspect of the general problem of developing more reliable programs. Program verification is desirable as it eliminates the time-consuming task of debugging programs, and guarantees that a verified program will always behave as it was intended to. Since proofs of program correctness can be very long and are themselves prone to human error, it is desirable to obtain machine assistance either to check the hand proofs or, if possible, to discover the proofs independently. The ultimate aim is for a programmer to be able to present his program together with its specifications to the computer which, as well as looking for syntax errors, will attempt with the programmer's help to verify the program, either certifying it correct or detecting any (semantic) errors, possibly by giving counter-examples. While such a system, for practical programs, still lies in the future, considerable progress has been made both in finding general methods of proving correctness (and other properties) of programs, and in mechanizing these methods.

In this thesis, we are mainly concerned with the following aspects of program verification:-

1) The use of alternative programs (*virtual programs*) as specifications of a given (or *actual*) program, and an associated inductive proof method.
2) The ability to easily add new knowledge about the different (mathematical) domains programs might operate on.
3) The design and use of a simple, interactive theorem prover to prove the verification conditions.
4) The use of a real language's compiler and normal run-time system to generate, by symbolic execution, verification conditions for programs written in that language.

The principal contribution of this thesis is twofold. First, it describes and discusses a new method of proving the partial correctness of programs. This method, *continuation induction*, is based on symbolic execution; allows recursive, iterative and non-deterministic programs to be treated uniformly; handles escapes and procedures with side-effects; and is especially convenient for proving properties of certain recursive programs. Secondly, the thesis describes an implemented program verifier which uses both this method and Floyd's method of inductive assertions. While the theorem-prover of our verifier is not as powerful as some others, the verifier is of interest for the way it uses high-level descriptions of programs, and for the nature

of the extensive interactive facilities provided. It also shows that non-trivial programs can be verified conveniently using a naive theorem prover with human assistance.

Using this method and the interactive theorem prover described, our system has verified, sometimes using human assistance, the following programs, among others:-

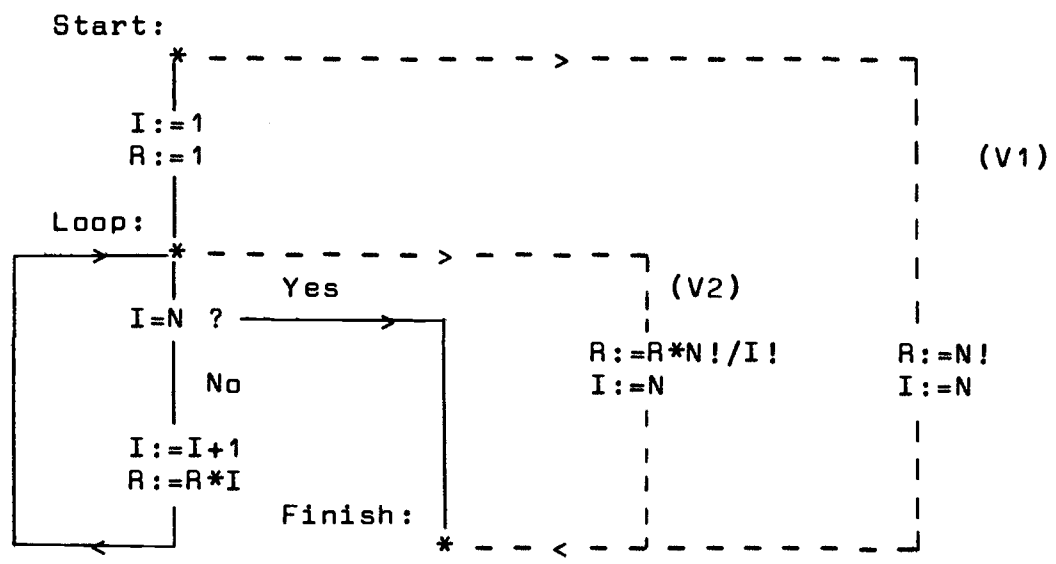
1) The 91-function,
2) A version of Ackermann's function,
3) Various common numerical programs,
4) A searching program which jumps out of a block,
5) Programs which test whether one array is equal to, or a subarray of, another,
6) Two simple sorting programs,
7) A program to invert a permutation "in place".

A list of programs verified by the system, together with their specifications, is given in Appendix 4.

The program verification system we have implemented verifies POP-2 programs, and is written in POP-2 (Burstall, Collins and Popplestone 1971), a language designed for use in artificial intelligence and combining features of both ALGOL and LISP. However the proof method used is applicable to any similar language. In our system, all the control aspects of symbolic execution are handled by the normal POP-2 run-time system. Thus, any correctness results proved are true with respect to the actual implementation of the language, rather than with respect to some abstract

definition of it.

To briefly illustrate our method of program verification, consider the following example.



The program in solid lines on the left is the actual program to be verified. The intention of this program is to set R to n! (the factorial of n) where n is the initial value of N. It also sets I to N. This is expressed by the virtual program (V1) in broken lines on the right: Start: - -> Finish. We wish to show that for all values of n the two programs return the same results, that is they are equivalent, provided the actual program terminates. Just as a loop must have an invariant in Floyd's method, so it must have a virtual program describing it in ours. The virtual program (V2) corresponding to the loop in the actual program is shown in broken lines, Loop - -> Finish, and we must also show that whenever the loop alone terminates it returns the same results as its virtual program.

Each of the actual programs (the whole program and the loop) is now shown separately to be "equivalent" to its corresponding virtual program. This is done by symbolically executing both the actual program and the corresponding virtual program from the same state, and checking that they terminate in the same state. Considering the inner pair of programs first, there are two pairs of paths:

```
1) Loop:N=n,I=i,R=r ——→ Finish:N=n,I=i,R=r with i=n

        versus

   Loop:N=n,I=i,R=r - -> Finish:N=n,I=n,R=r*n!/i!

2) Loop:N=n,I=i,R=r ——→ Loop:N=n,I=i+1,R=r*(i+1)
                              with NOT i=n
                    - -> Finish:N=n,I=n,R=r*(i+1)*n!/(i+1)
                              with NOT i=n
                                 (induction hypothesis)
        versus

   Loop:N=n,I=i,R=r - -> Finish:N=n,I=n,R=r*n!/i!
```

Clearly the resulting state vectors are the same in each case. Notice how we used the induction hypothesis that the two programs are in fact equivalent by executing the virtual program when we returned to Loop. The proof is thus by induction on the number of times the program goes around the loop. To verify the complete program we must compare the following pair of paths:

```
3) Start:N=n,I=i,R=r ——→ Loop:N=n,I=1,R=1
                     - -> Finish:N=n,I=n,R=1*n!/1!
        versus

   Start:N=n,I=i,R=r - -> Finish:N=n,I=i,R=n!
```

Again the resulting state vectors are clearly the same. In this case we used the above result that the actual and virtual programs from Loop are "equivalent" to circumvent

the loop by executing the virtual program from that point.

The method can be applied to recursive procedures in the obvious way by giving a virtual program to describe the body of the procedure. However its utility comes from the fact that the virtual programs may be considerably more complex than they were here. In particular, they may include conditionals, jumps and non-deterministic operations.

Generally, as in this example, the virtual program which describes an actual program cannot be expressed using only the primitives of the programming language, even though the actual program is computing some well-known mathematical function such as the greatest common divisor of two numbers, the factorial of a number, the inverse of a matrix, the product of two matrices, the transitive closure of a relation, the inverse of a permutation, the connected components of a graph, etc. It is clearly desirable in such cases that the specifications of the actual program should be in terms of these mathematical concepts rather than at the level of the representation used by the actual program. Our system allows the user to declare such functions as new (undefined) primitives of the programming language and use them to write his specifications for the program. The properties of these functions can be defined by giving axioms and rules which are used in actually verifying the program. These rules will also relate the abstract mathematical objects involved to the data structures used to

represent them. (A complete list of such rules used is given in Appendices 2 and 3.) The verification system can then be easily extended to deal with programs operating on a variety of mathematical domains.

The intrinsic complexity of such a domain is often the cause of any difficulty in verifying (or understanding) a given program. To verify such a program requires not merely a knowledge of programs and programming languages, but also the ability to prove theorems within the theory of this domain. Since mechanizing such proofs is a substantial research problem in its own right, we believe it should be separated as much as possible from the task of program verification. We have done this by building an interactive system which can do some theorem proving by itself, but which accepts new information about the current domain from the user (i.e. the programmer) when it is required. This new information will usually consist of rules describing the mathematical functions used in the program's specification. Of these rules, some are treated as definitions and are accepted without question; for a complete proof the others should be shown to follow from the definitions, but they can be assumed and their proofs postponed. The theorem prover will then attempt to apply these new rules in its continued search for a proof. If it still fails, the user can direct the proof process himself by providing lemmas, instantiating variables, adding new hypotheses and so on. A similiar view of program verification is taken by Good, London and Bledsoe (1974).

As mentioned above, program verification is only one approach to the broader goal of developing better, more reliable programs. One important alternative is the idea of "structured programming", a technique for developing a program in a systematic way and possibly generating a proof of the program's correctness at the same time. This often involves programming at different levels, implementing a program (primitive) at a higher level by means of a lower level program. In the long run, we believe that "structured programming" and better education of programmers will be the best way to improve the quality of programs.

Two other approaches related to program verification are program transformation and program synthesis. By "program transformation" we mean the process of changing a simply-stated program at the source language level to make it more efficient. Program synthesis is the problem of producing from the (possibly incomplete) specifications of a program a program which satisfies them. We believe that an interactive approach is the best one for each of these three tasks and that since they all involve reasoning about programs, progress in any one field should be applicable to the others.

The organization of this thesis is as follows: The next section presents a review of previous work done on program verification; it describes both proof methods and implementations of these methods. The essence of the thesis is in Chapters 2 and 4. An overview could be obtained by

reading these chapters and skimming through Chapter 6. Chapter 2 describes our proof method in detail; Chapter 3 gives some applications of the method and compares it with various other methods; Chapter 4 describes the verification system we have implemented; in Chapter 5 the algebraic manipulation system, automatic theorem prover, and interactive facilities are described; Chapter 6 presents and explains the behaviour of our verification system on some typical examples; and Chapter 7 presents our conclusions and suggestions for future research.

1.2 Related work

There are four commonly used inductive methods for proving properties of programs. These are: inductive assertions (Naur 1966, and Floyd 1967), recursion induction (McCarthy 1963), computational induction (Park 1969, and deBakker and Scott 1969), and structural induction (Burstall 1969). Of these, inductive assertions has been the principal one concerned with iterative programs and assignments, while the remainder have dealt mainly with recursive functions. Whereas recursion induction is used to prove the equivalence of two programs, the other methods are usually used to prove properties (or correctness) of particular programs. The first method is essentially induction on the length of the computation path, the second and third are induction on the depth of function calls, and the fourth is induction on the data structures being manipulated. We shall look briefly at

each of these methods in turn, and then describe some implemented systems for proving properties of programs which use them.

The most commonly used inductive method, especially in automatic verification systems, is that of Floyd. In this approach, assertions (about the values of the program's variables) are attached to key points (such as loops) in the program, and an assertion must be true each time control passes through the relevant point during the program's execution. In particular, an assertion is usually attached at the end of the program. Verifying the correctness of the program consists of proving that for each path through the diagram, each assertion implies the next one in the path provided the effects of the intervening program statements are taken into account; such implications are called verification conditions. Manna (1969) describes a similar method which may be thought off as attaching assertions to points such that the assertion is true during some pass through the point, rather than all passes as in Floyd's method. In Manna and Pnueli (1970) this method has been generalized to handle recursive functions. Manna's method was further modified in Burstall (1974). Hoare (1969) described a structuring principle for using Floyd's method which has since become widely accepted.

Recursion induction is used to prove the equivalence of recursive functions. To prove the equivalence of two functions f1 and f2 over some domain A, that is, that

f1(x)=f2(x) for all x in A, it is necessary to find a third recursive function g such that both f1 and f2 satisfy the defining equation of g and g is defined for all x in A. Although this was perhaps the first method explicitly suggested for proving properties of programs, it seems to have been little used in practice. McCarthy (1962) showed that recursion induction could also be used to prove the equivalence of iterative programs.

To explain computational induction, assume for the moment that UU is the totally undefined function, that T is a continuous functional and that F1 is the function defined by F <= T[F] (these terms are explained in Manna et al. 1972). Then to prove the property P(F1) of F1, it is sufficient to check that P is true before starting the computation (P(UU)), and show that if P is true at one stage of the computation, it remains true after the next step (P(F) implies P(T[F]) for every F). Morris (1971) described a variant of this called truncation induction, which bears the same relationship to computational induction as course-of-values induction does to ordinary mathematical induction.

The final method is structural induction which is described in Burstall (1969) and was first used (in computing) by McCarthy and Painter (1967) who proved a compiler for arithmetic expressions correct. It is applicable to any class of finite structures (often called "records" or "plexes") built up from a set of atoms, and

which do not contain circularities. The induction principle is: If for some set of structures a structure has a certain property whenever all its proper constituents have that property, then all the structures in the set have the property. Logicians frequently use structural induction to establish meta-theorems, by inducting upon the structure of formulas in the theory.

Each of these methods has its own advantages and disadvantages; the question of which to use is largely a pragmatic one. For instance, in the presence of assignments to data structures, Floyd's method is applicable whereas structural induction is not. The paper by Manna, Ness and Vuillemin (1972) is a very readable introduction to the various inductive methods, and has many examples of their use.

We shall now review other implemented systems, paying particular attention to the aims, methods and accomplishments of the system, and the features of the languages used to present programs and assertions (or theorems). Almost all these systems are concerned with the inductive assertions method for flow-diagram languages; the exceptions are Milner (1972) and Boyer and Moore (1973). These two, together with those of Deutsch; Igarashi, London and Luckham; Waldinger and Levitt; and Good, London and Bledsoe are the most powerful of the systems. Our system is most closely related to those of King, Deutsch, Waldinger and Levitt, and Good et al.

The first of these systems, and in many ways the prototype for several others, was that of King (1969). King's system dealt with an ALGOL-like flow-diagram language by Floyd's method. He allowed integer-valued variables and one-dimensional arrays with integer elements. Statements included assignments to variables and array elements, conditional statements, and goto statements. Procedure calls were not allowed. The system was designed as a fully automatic implementation of Floyd's method. The user submitted his program text with assertions, and King's system then generated the verification conditions and tried to prove them. The proof was done by an arithmetic theorem prover designed specifically for the task. Several interesting programs were thus verified, including an array sorting program, and a program to raise an integer to a power using the binary representation of the power.

All the knowledge available to King's system was already built-in. Assertions were just boolean expressions with universal and/or existential quantifiers. It was not possible to add a procedure to express an assertion, which severely limited the expressive power of the language. For example, functions such as summation and greatest common divisor were not built-in, and thus not available. The system was written in assembly language using macros and was very fast.

Good's thesis (1970) described another formalism for programs and proved several results within that formalism. He also presented a program, operating on a language similar to King's, which generated verification conditions from the user-supplied text and assertions. However it made no attempt to prove them, providing only a book-keeping service to the user. Because the program was not designed to "understand" the assertions, the assertion language was very flexible, consisting of arbitrary text strings in which occurrences of program variables could be recognized and substituted for. One contribution of Good's work was to show that in the presence of subscripted variables (array elements), generating verification conditions by working forward along a path, from the initial assertion to the final assertion, generated simpler conditions than working backwards. King's system, which worked backwards as Floyd had originally suggested, was restricted by the large number of cases sometimes generated.

Cooper (1971) presented a theorem prover which dealt with flow-diagram languages like those above, without provision for arrays. The program was designed to automatically generate and prove the termination and correctness conditions for flow-diagrams using the Presburger algorithm (for arithmetic without multiplication), but was limited in its scope.

In 1973, accounts of several impressive systems appeared: notably Igarashi, London and Luckham (1973), Deutsch (1973), Waldinger and Levitt (1973), and Boyer and Moore (1973). Each of these is significantly more powerful than those described above.

Igarashi et al. apply Hoare's method to generate verification conditions for programs written in the programming language PASCAL (Wirth 1971). The verification conditions are then given to a resolution theorem prover to be proved. An example of a program they can verify is Hoare's (1971b) FIND. Their system does very little actual resolution and a lot of simplification and reasoning about equality.

Deutsch's system is a straight-forward application of Floyd's method: it takes a program with assertions, generates the verification conditions, and uses an automatic theorem prover to prove them. Programs are written in essentially the same language that was used by King; procedure calls are not allowed. The assertion language consists of quantified boolean expressions, but also allows parameterized assertions, defined by one-line boolean procedures. The system, like King's, stores all expressions in canonical form. The increased power comes not so much from a better theorem prover, but from the simplifications and deductions made while generating the verification conditions (which is done by forward substitution), and from the context mechanism used. Interactive facilities which

enable the user to help the system find a proof are provided, but no examples of their use are shown. The system is written in LISP and is rather slower than King's system; it can verify all of King's examples, FIND, and a complex enumerative program, among others.

Waldinger and Levitt (1973) have implemented a flexible system in QA4 (Rulifson et al. 1972), a very high level language designed for use in artificial intelligence. The programming language considered contains integers, reals, arrays, lists, conditionals, assignments, and recursive procedures. The assertion language is QA4 itself (slightly extended) which gives maximal flexibility to use arbitrary functions and predicates which describe the program's properties. For iterative programs Floyd's method (with backward substitution) is used to generate the verification conditions, while for recursive programs the Manna-Pnueli method (computational induction) is used. Their theorem prover is a set of QA4 functions. The beauty of their system is in the simplicity of the functions (or rules) making up the theorem prover, the ease with which new information, in the form of new rules, can be added, and the natural flavour of the resulting proofs. However because of the backtracking and pattern matching involved in running QA4 programs the system is very slow, and the interactive facilities available are very limited. The system can verify all of King's examples, FIND, and a version of the unification algorithm, among others. It is intended that the knowledge about programs embedded in the system will

subsequently be used in automatic program modifcation and synthesis systems.

In a recent report (Good, London and Bledsoe 1974), a philosophy and system quite similiar to ours is described. The authors have used a sophisticated algebra system (REDUCE, Hearn 1971) to combine Bledsoe's interactive theorem prover (see below) with a verification condition generator derived from that of Igarashi et. al. (1973) to produce an interactive verification system for PASCAL programs. Their system is still in a state of development and appears very promising.

Suzuki (1974) describes the simplification and logical reduction rules used in an extension of the Igarashi, London and Luckham verifier. This system also uses high-level, user-introduced predicates with axioms describing their properties. It is completely automatic and can prove Hoare's FIND and Floyd's (1964) TREESORT programs. Von Henke and Luckham (1974) use this system to describe a methodology for verifying programs. The methodology involves using information from attempted verifications to successively refine and modify both program and assertions until they can be shown to be consistent.

Two other systems are perhaps worth noting: Gerhart (1972) describes the use of Floyd's method to prove properties of programs in a subset of APL, but the only process automated is the verification of the compatibility of argument types and APL operators. Ragland (1973)

describes a program written in a simple language called NUCLEUS which generates verification conditions for NUCLEUS programs, and he gives a (hand) proof of this particular program's correctness.

All the above systems have required that the inductive assertions be supplied along with the program. Although it has been suggested that a programmer does not really understand his program until he can supply these assertions, their discovery can be quite difficult. Some interesting work on heuristically automating this process has been done by Elspas (1972), Katz and Manna (1973), Wegbreit (1973) and German and Wegbreit (1975).

A different approach has been taken by Sites (1974) who has devised a method for showing that a given program (without assertions) always terminates without overflow or array subscript errors. The method has been used on some difficult programs but fails when the termination proof depends on non-trivial mathematical facts. It has not been mechanized.

Boyer and Moore (1973) have written a program which can prove difficult theorems about pure LISP programs by structural induction, and which is based on the idea of symbolic execution. It is described in detail in Moore (1973). The program uses LISP as both its logical and assertion languages, allowing arbitrary LISP functions to be used as assertions. The main achievements of the program are that it requires no assertions other than the one to be

proved; it generates its own induction hypotheses; and it generalizes the theorem to be proved in an intelligent way whenever necessary. For instance, given only the LISP definitions of SORT and ORDERED, the program can prove that (ORDERED(SORT L)) is true. A weakness of the program is that it cannot deal with programs which "recurse up" rather than "down" (iterative programs), let alone with destructive assignments to data structures.

Milner's LCF proof-checker (Milner 1972) is in a sense the most powerful of all, albeit that the power is entirely controlled and directed by the user. This system is an implementation of Scott's Logic for Computable Functions (Scott 1969). The basic induction rule is computational induction, although a proof of the recursion induction rule is presented as an example. The program is an LCF proof-checker. The syntax and semantics of a wide range of programming languages may be expressed in LCF, including recursive programs and programs which have other programs as arguments and results. The program accepts expressions in LCF as theorems to be proved, and then obeys commands from the user directing the application of the rules of inference. A powerful simplification routine shortens the proofs, but they can still be very long. The program keeps track of the goals to be established and the steps carried out in each proof. In Milner and Weyhrauch (1972), the authors describe the use of the LCF proof-checker to verify the correctness of a compiler. Newey (1975) has used the LCF proof-checker to prove various theorems about LISP

functions, including the "correctness" of the LISP interpreter.

Since program verification inevitably requires a large number of theorems to be proved, any other research done on automatic theorem proving should be of interest. However we shall mention only one such system: that of Bledsoe and Bruell (1973). They describe an interactive theorem prover for general topology which works in a fairly natural manner by subgoaling, applying rewrite rules and definitions, and using special heuristics for topology. We have borrowed ideas from them about organization and interactive facilities for our own system.

We shall compare our own work with many of the above proof methods and verification systems in more detail later, when we discuss particular aspects of our proof method and verifier.

1.3 Notation and conventions

We have tried to use standard mathematical and programming notation throughout. When talking about programs and their executions we distinguish between upper cases letters which refer to identifiers and lower case letters which refer to logical variables (their values). Words being defined or emphasized are underlined, and words used in an unusual way are enclosed in quotes (") as usual. For syntax definitions we use BNF (Backus-Naur Form) as

described in the POP-2 reference manual.

Because of the restrictions of preparing this document on the computer we have written e.g. x1, x2, ..., xn instead of using subscripts, and have used the following abbreviations.

Symbol	Stands for	Meaning
&	$\wedge$	logical conjunction
v	$\vee$	" disjunction
=>	$\supset$	" implication
<=>	$\equiv$	" equivalence
NOT	$\neg$	" negation
FA	$\forall$	for all, universal quantifier
EX	$\exists$	there exists, existential quantifier
EPS	ε	Hilbert's epsilon symbol
==	$\equiv$	identity (of expressions), equivalence (of programs)

Chapter 2. THE INDUCTIVE PROOF METHOD: DESCRIPTION

2.1 Introduction

As we have already indicated, the proof method we shall present relies on the use of alternative programs as specifications. We originally started studying this idea and its consequences as it seemed to lend itself well to the idea of constructing proofs by symbolic execution (Topor and Burstall 1973) independently of Deutsch (1973) and others who have used symbolic execution in systems based on inductive assertions. The main advantages of the proof method we eventually developed are that programs containing iteration, recursion or non-determinism are all treated in the same way, and that it can handle escapes and procedures with side-effects. Moreover the method is equivalent to computational induction (Manna and Pnueli 1970) for recursive programs, is essentially a generalization of Floyd's method of inductive assertions (Floyd 1967) for iterative programs, and can easily be extended to yield termination proofs in the same way that Floyd's method can. We refer to the method as continuation induction since it involves providing a "continuation" for certain points in the program, that is, a function (or relation) computed by the program from that point until its end. The work in this chapter, with the exception of Section 2.6, was done jointly with Dr R.M. Burstall.

In this chapter we shall try to explain the method in detail independently of any particular implementation. But before going on to this we shall define some of the terms we shall be using. First, we use the word "program" in a very broad sense throughout to include any sequences of statements possibly preceded by one or more function definitions. An actual program is simply a program whose correctness we are currently trying to prove; since the correctness of a program depends on the correctness of its constituents, these constituents will also be considered as actual programs at times. A virtual program is the program which serves as the specification of an actual program, and to which the actual program is to be proved "equivalent". We shall describe later just what the nature of these programs may be.

But what do we mean by saying that two programs are "equivalent"? There are (at least) three possible answers. The first is that two programs are said to be (strongly) equivalent if for all inputs either they both terminate and produce the same results, or neither terminates, that is they are equal as partial functions. This concept is too strong for our purposes since although virtual programs always terminate, actual programs may not. The second possibility is that of weak equivalence: two programs are said to be weakly equivalent if whenever they both terminate they produce the same results. Any two programs are weakly equivalent to a program which never terminates. Finally, a program P is said to be included in a program Q, written

$P \sqsubseteq Q$, if whenever P terminates so does Q and they produce the same results, that is, P is less defined than Q as partial functions. Since we are trying to show that whenever the actual program terminates it produces the same results as the virtual program (which always terminates), this is the desired concept. Notice that inclusion is a reflexive and transitive relation, but is not symmetric.

We can now define a program to be partially correct if it is included in its corresponding virtual program; it is totally correct (i.e. is partially correct and terminates) if it is strongly equivalent to its virtual program.

2.2 Actual programs

In this section we shall describe the type of programs to which the proof method is applicable, how programs are executed, and what is really meant by "symbolic execution". This lays the groundwork for the following sections in which we shall describe how to give specifications for actual programs in terms of virtual programs, and how to construct proofs from such pairs of programs.

2.2.1 Nature of amenable programs

One of the aspects of continuation induction is the uniformity with which various control features of programming languages can be handled. In particular, functional and imperative programs are treated in almost the

same way. The method is applicable to programs constructed from the following control features:

assignments, statement sequencing, blocks, conditional statements and expressions, labels, arbitrary jumps, while-statements, procedures, escapes and functional arguments.

While we do not believe programmers should use jumps indiscriminately, programs using such jumps can be handled. Jumps out of blocks and escape functions constructed using Landin's J-operator (Landin 1965) are also permitted provided they are not used to jump back into functions. Escapes are functions which jump out of their defining function when called, and are mainly used for error trapping and to avoid "unwinding" recursive function applications. Both recursive procedures and procedures with side-effects are allowed. Certain types of functional arguments are allowed, but we do not yet know how to deal with label arguments.

The only restrictions on the data types allowed come from the formal system one has available for reasoning about the data. In the verification system which we have implemented, the data types used are truth values, integers, one-dimensional arrays of integers, and lists (without destructive updating). The only barrier to introducing other data types such as reals, strings, records etc. would be the need for an extended algebraic and inference system to manipulate and reason about them.

Rather than specifying the syntax and semantics of some particular programming language and then describing how to verify programs written in that language, we shall try to describe the proof method in a language independent fashion. Since the proof method is defined in terms of the semantics of the language being considered we shall actually be describing a family of proof methods. Thus the user should instantiate our general description which follows to the particular language and formal (operational) semantics of his choice.

Unfortunately, it is necessary to use some particular language while describing the method. We shall use an ALGOL-like language in which functions are defined, for example, by abs(X) = <u>if</u> X>0 <u>then</u> X <u>else</u> -X, in which statements are grouped into blocks by parentheses, and in which the function <u>return</u> returns its argument as the result of the innermost function in which it appears. In all the examples we shall give, parameters are passed by value, but this is not a restriction on the proof method as call by reference and call by value-result can also be handled.

2.2.2 <u>Execution of programs</u>

Since the proof method is based on the idea of symbolic execution of programs, we start by describing ordinary execution. To do this it is necessary to define a machine state and say how the various commands of a program affect this state. However, we shall merely give an outline of the

operational semantics for the language being considered. When actually doing proofs the user would use the formal operational semantics of his particular language, though in Chapter 4 we explain how this can be made unnecessary.

Consider first a flow diagram type program without any procedure calls. Then it can be seen that the machine state is specified by:

1) a state vector which is a mapping from identifiers to values, and
2) an instruction pointer which indicates the program statement about to be executed.

For example, a typical machine state might be

Start: I->0, N->10, R->1, A-> <array [1:4] 3 5 2 4> ,

where Start is the instruction pointer (or label), and the value of A is an array of length 4 with the values shown. In such a case it is obvious how the various commands - assignments, tests, and jumps - affect the state, so we do not give the corresponding state transformations.

However, when we introduce procedures into the language the situation becomes more complicated, especially since we want to allow recursive procedures. It is necessary then to introduce a third component into the machine state:

3) a control stack which holds the local variable values and return addresses necessary to implement procedure calls and returns.

This corresponds to the Dump of Landin's SECD machine (Landin 1964). Calling a procedure now involves pushing an activtion record corresponding to the new procedure onto the control stack, setting the instruction pointer to the start of the procedure body, and adjusting the state vector (or environment) by associating the actual and formal parameters of the procedure. Returning from a procedure involves "undoing" these changes. We will not bother to invent details of how this could be done in our pedagogical language. Notice however that the semantics of the goto-statement is now more complex, since jumping to a non-local label will involve popping the control stack and changing the state vector.

We refer to the sequence of points in the program through which the program passes as the computation path. When operating on real data, a deterministic program follows just one computation path as all the tests can be evaluated. This is not the case for symbolic execution which we discuss next.

However, before doing so, we remark on another way to describe the semantics of our programming language. This is to systematically transform each program in the language into a system of recursive equtions and then use one of the evaluation rules described in Manna et al (1972), for instance, to evaluate the program for particular input. This is the method we shall actually use in Section 2.6 when justifying the validity of this proof method, so the reader

should keep it in mind.

2.2.3 Symbolic execution of programs

To prove properties of a program it is necessary to determine the program's behaviour not only on particular input data, but on all possible data. One way to do this is to run the program with an initial state vector which represents all possible state vectors. In this case the values of the program identifiers are symbolic expressions constructed from variables (Skolem constants) which are specified to represent particular data types. These variables are considered to simultaneously take all possible values of the appropriate type. A state vector which maps program identifiers into such a domain of symbolic algebraic values is called a symbolic state vector. A typical symbolic state vector might be:

(X->x+1, Y->2*y, A->a0).

We shall occasionally refer to a general symbolic state vector which is just a symbolic state vector of the form

(X->x, Y->y, ...)

where X, Y, ... are all the identifiers of the program and x, y, ... are simply logical variables with the same names as the identifiers.

The fact that the initial input to the program may not be quite arbitrary, but may be required to satisfy certain conditions, x>=0 say, can be represented by introducing into the state a new component called a path condition which is a

logical expression expressing these conditions. This extra component is also very important for deciding what to do at tests as will be seen below. Thus a symbolic machine state has the following components:

1) a symbolic state vector (sv),
2) an instruction pointer (ip),
3) a control stack (cs),
4) a path condition (pc).

When writing out a state we will tend to omit those components which are not of immediate interest and to refer to the components of a state by the abbreviations shown above.

The process of running a program from such an initial state is called symbolic execution. There are two obvious difficulties. The first is that standard operations such as +, *, =, <, cons, hd, etc. cannot be applied since their arguments are variables rather than numbers or lists and are hence of the wrong type. The solution is to redefine these operations to be symbolic ones, that is operations which construct new (symbolic) terms from their arguments, possibly simplifying the result. Functional arguments are dealt with similiarly. The second difficulty is that the truth of tests in conditional statements can no longer always be determined - at least, not by evaluation alone. If it is possible to prove from the current path condition that the test must be true, or that the test must be false, then the computation simply proceeds along the appropriate

path. Otherwise the computation path splits into two new ones. On one the test is assumed to be true and is added to the path condition; on the other it is assumed to be false and its negation is added to the path condition.

We now describe this process more precisely in the following algorithmic way. In this description we assume that all conditionals are brought to the top level, e.g., "f(if P then s1 else s2)" becomes "if P then f(s1) else f(s2)", and hence "symbolic evaluation" is simply the process of constructing symbolic terms. The "with - do" construct allows us to refer to the components of a structure (in this case a state) by the names of their selectors, and the function mkS constructs a state from its components. We assume there is an operation advance which moves the instruction pointer on to the next instruction (unless the previous instruction was a goto or some other statement which affects the instruction pointer), and that there is a function sym-val which symbolically evaluates an expression (possibly performing side-effects on SV as well).

```
Sym-exec(STATE) =
  with STATE do
  repeat (
if IP is at "halt" then return({STATE})
else if IP is an assignment (LS := RS)
     then (SV := SV[sym-val(RS)/LS]; advance IP)
else if IP is a conditional (if P then s1 else s2)
     then if PC => P then IP := s1
          else if PC => NOT P then IP := s2
          else return(Sym-exec(mkS(SV,s1,CS,PC & P))
              UNION Sym-exec(mkS(SV,s2,CS,PC & NOT P) )
else (execute instruction normally; advance IP) )
```

Figure 2.1 - Symbolic execution

Thus symbolic execution is conceptually like normal execution in all respects except for functional calls, conditionals and the components of the machine state. Handling these as described above, it generates a tree of computation paths, each path having an associated machine state which changes as the execution progresses, and returns the set of states at the ends of the paths. Of course, if performed on a program with loops or recursion, this process would continue indefinitely; we will explain shortly the induction principle used to prevent this. The idea of symbolic execution, at least with regard to the symbolic evaluation of expressions, was first used by Perlis and Iturriaga (1964). It has also been used by Darlington (1973) in program optimization, by Deutsch (1973) and Boyer and Moore (1973) in program verification, and recently by Boyer, Elspas and Levitt (1975) and King (1975) in program testing.

2.3 Program specification: virtual programs

2.3.1 Virtual programs

Clearly, for a virtual program to be acceptable as the specification of the actual program the virtual program must itself be clear, precise and unambiguous. To achieve this, the virtual program is written in the same language as the actual program, subject only to the following condition:

> The virtual program may not contain any loops, backward jumps or recursive calls. Moreover any subroutines called by the virtual program must also satisfy this restriction.

This restriction ensures that the virtual program will always terminate, and will in fact execute each of its statements at most once. With this restriction it should be so obvious what the virtual program does that it is acceptable as a specification for the actual program. Note that the virtual program may use any other features of the programming language including conditional statements and (forward) jumps out of blocks. It may also include jumps to labels in the program surrounding the current actual program.

2.3.2 Language extensions

Although virtual programs can sometimes be written in the language without any extensions (e.g. the 91-function of Section 2.5.1) it is usually necessary to introduce new functions into the language. In writing the virtual program we can use any mathematical functions, such as factorial, which are appropriate for the domain on which the actual program is operating, even if they are not provided as procedures of the language. To do this, it is necessary to be able to declare the mathematical function as an operation of the language which simply constructs a symbolic term, and to provide a definition of the function which can be used when proving the inclusion of the two programs. Frequently, as for factorial, this definition will be in the form of a

set of recursive equations, reducing the correctness problem to that of showing the equivalence of the iterative and recursive definitions of the functions. Other ways of defining these new functions can also be used; the only restriction is on the inference system available. The main purpose these functions serve is to provide some form of "canned loop"; this is specially obvious for arrays where we will need to describe the effect of a program on some portion of an array without actually using any loops. For example, writing A[I,J] for the sequence A[I], A[I+1], ..., A[J] of array elements, the function eqseq could be introduced to test the equality of two array segments: eqseq(A[I,J],B[I,J]).

Sometimes, however, the mathematical functions required to express what a program does become more complicated than the program seems to deserve. For instance, consider the program:

while X<A do X:=2*X;

Assuming A>0 & X>0, the corresponding virtual program is:

X := 2^(ceiling(log(A/X)))*X

where the logarithm is to the base 2, and ceiling(x) is the smallest integer not less than x. However this program seems more complicated than the original one, and more likely to contain errors. A clearer way to write the virtual program is

X := 2^(MU Y)(2^Y*X>=A)*X

where (MU y)P(y) is the least integer satisfying P(y). Alternatively, if it is not important that X is assigned the

least y satisfying 2^y*x>=a, we could use Hilbert's epsilon operator and write

```
X := 2^(EPS Y)(2^Y*X>=A) * X;
```

In fact, we shall use only this operator throughout our discussion (in addition to the usual universal and existential quantifiers). For Hilbert, (EPS y)P(y) denotes "any object y satisfying P(y), or anything if no such object exists". Thus EPS can be used to construct arbitrary terms for use in virtual programs. It can be used to produce lists and arrays satisfying certain conditions as well as numbers; the type of the resulting term will always be clear from the context. Thus (EPS x)(0=<x & x=<y) could be any of 0,1, ..., y-1,y (assuming integer arithmetic is being used). Note that EPS is really a non-deterministic operator so that (EPS x)P(x) = (EPS x)P(x) is not a valid formula. We have extended EPS to produce tuples of objects, e.g. (EPS x,y,z)P(x,y,z) denotes any tuple (x,y,z) such that P(x,y,z) or any tuple if no such tuple exists. Also in this work, unlike Hilbert, we consider the term (EPS x)P(x) as referring to the set {x: P(x)} . The epsilon operator was originally introduced by Hilbert in his study of mathematical logic; its use is described in Leisenring (1969) . It is especially useful in programs dealing with arrays. For example, the virtual program corresponding to the outer loop of a sorting program might be:

```
A := (EPS A1)(perm(A[1,I],A1[1,I]) &
                (FA J)(1=<J<I => A1[J]=<A1[I])).
```

More detailed examples will be given later.

2.3.3 Inclusion statements

A key concept in this proof technique is that of the inclusion statement. This is a statement associating each actual program with its corresponding virtual program. An inclusion statement has the following components (with their abbreviations in parentheses):

1) The actual program (ap),
2) The virtual program (vp),
3) The preconditions under which ap $\sqsubseteq$ vp (prec),
4) The starting point (sp),
5) All the possible end points (ep),
6) The variables w.r.t. which ap $\sqsubseteq$ vp holds (vs).

Clearly, representations of the two programs are required. The preconditions are mainly required to ensure that the virtual program always terminates, though by making them sufficiently detailed and correspondingly simplifying the virtual program, they can be used in what is effectively a proof by inductive assertions. This will be illustrated later. The start and end points define the scope of the inclusion statement. This is important as the actual program may be contained in a larger program. In this case the actual program itself can be omitted from the inclusion statement as it is effectively specified by the description of its scope. For reasons that will become clear later, it is also necessary to state whether an end point corresponds to a normal exit from a recursive procedure or not. For example, if Ret is an escape function in a recursive

procedure Search the virtual program corresponding to Search would have the form:

```
In:  s1;
     if "condition" then (s2; Ret()) else s3;
Out:
```

which has two exits: one recursive one at Out, and one non-recursive one at Ret. A detailed example of this nature will be given in Section 2.5. While such escapes are the main motivation for multiple exits, they can also be useful in flow-chart programs which jump out of loops, as will be illustrated later. Finally, the last component serves to restrict the amount of testing to be done: variables which are not used outside the scope of the inclusion need only be tested for inclusion if particularly desired. The value of a function and result parameters of procedures must always be tested. Such an inclusion statement asserts:

> Under the given preconditions, the actual program which starts at the given starting point and is bounded by the given end points is included in the corresponding virtual program with respect to the given variables.

That is, if the actual program was replaced textually by the virtual program, the result would be either the same or possibly more defined.

2.3.4 Specified programs

Now, how do we give specifications for a complete program? Obviously, we must first give an inclusion statement for the complete program. In addition, it is simply necessary to provide a separate inclusion statement for each loop or function used in the program to be

verified. More precisely, in a flow-chart program, each cycle must contain the start of at least one inclusion statement, and every program and subroutine must contain an inclusion statement at its beginning. For example, two inclusion statements were required in the factorial program of Chapter 1: one starting at Start and ending at Finish, and one starting at Loop and ending at Finish.

2.4 General description of method

Having described how to give the specifications in a form appropriate for this method, we now say what the method actually is. We do this in two stages, first for individual inclusion statements and then for complete programs.

2.4.1 Individual inclusion statements

Consider an inclusion statement with actual program A, virtual program V, preconditions C, start point s, end points e1,...,en, and variable list L. The theorem expressing the correctness of the inclusion statement is:

> "For all integers k>=0, if C is true, and if A returns to s k times before terminating at an ei, then A $\sqsubseteq$ V with respect to the variables in L."

It is proved by induction on k, that is, on the length of the computation path.

The basic idea is simply to symbolically execute both A and V and to compare the results. In each case we start at s with a general symbolic state vector and with path condition C. Symbolically executing the virtual program is done straightforwardly as described above. If the execution reaches one of the end points, ei, that path of the computation is terminated, and the state at that point saved. The states at the ends of the computation paths are accumulated in a set, SVIRT say.

Symbolically executing the actual program is a little more complicated. As for the virtual program we start at s with a general symbolic state vector and path condition C, and start to execute the program. If one of the end points is reached, the state at that point is saved as before. If the program returns to s it attempts to prove the preconditions C using the current values of the program's variables and the current path condition. If this cannot be done there is an error either in the program or in the *inclusion statement*. *Otherwise the program* enters "hypothetical" mode and starts to symbolically execute the virtual program V . This corresponds to the application of the induction hypothesis that A $\sqsubseteq$ V. The behaviour at an end point ei is now more complicated. If ei is a normal exit from a recursive function, the program leaves "hypothetical" mode and continues to be executed. (This will only happen if the actual program A corresponds to the body of the recursive function.) Otherwise, if ei corresponds to an exit from a loop or to an escape exit, the

state is saved and the path terminated as before. The states at the ends of the computation paths of the actual program are also accumulated into a set.

We describe this process more formally in Figure 2.2, recalling that the components of an inclusion statement are ap, vp, prec, sp, ep and vs. Note how the sets of states are formed one member at a time, by side-effects, rather than by explicit unions as in Figure 2.1. In the actual implementation of the proof procedure these sets are formed without side-effects but coroutines are used instead.

```
Verify-inclusion-statement(INCL-STAT) =
with INCL-STAT do (
  STATE := mkS(general-symbolic-state-vector,  SP,
               arbitrary-control-stack,  PREC);
  SACT := {};
  Ex-act(STATE,INCL-STAT,false);
  SVIRT := {};
  Ex-virt(STATE,INCL-STAT,false);
  Compare(SACT,SVIRT) )


Ex-act(STATE, INCL-STAT, FROM-VIRT) =
with STATE do
with INCL-STAT do
repeat (
  if IP is in EP and not(FROM-VIRT) then
     (add STATE to SACT; return);
  FROM-VIRT := false;
  if IP = SP (other than initially) then
     (Check that PC => SV(PREC);
      Ex-virt(STATE,INCL-STAT,true);  return)
  elseif IP is an assignment (LS := RS) then
     (SV := SV[sym-val(RS)/LS]; advance IP)
  elseif IP is a conditional (if P then s1 else s2) then
     if PC => P then IP := s1
     elseif PC => NOT P then IP := s2
     else (Ex-act(mkS(SV,s1,CS,PC & P),INCL-STAT,false);
           Ex-act(mkS(SV,s2,CS,PC&NOT P),INCL-STAT,false);
           return)
  else (execute instruction normally; advance IP) )


Ex-virt(STATE,INCL-STAT,HYPMODE) =
with STATE do
with INCL-STAT do
repeat (
  if IP (=ei) is in EP and HYPMODE
  and the ei is "recursive" then
     (Ex-act(STATE,INCL-STAT,true); return)
  elseif IP is in EP and HYPMODE then
     (add STATE to SACT; return)
  elseif IP is in EP then (add STATE to SVIRT; return)
  elseif IP is an assignment (LS := RS) then
     (SV := SV[sym-val(RS)/LS]; advance IP)
elseif IP is a conditional (if P then s1 else s2) then
   if PC => P then IP := s1
   else if PC => NOT P then IP := s2
   else (Ex-virt(mkS(SV,s1,CS,PC&P),INCL-STAT,false);
         Ex-virt(mkS(SV,s2,CS,PC&NOT P),INCL-STAT,false);
         return)
else (execute instruction normally; advance IP) )
```

Figure 2.2 - Verifying one inclusion statement

We are now left with the problem of comparing SACT and SVIRT, that is, of showing that SACT $\sqsubseteq$ SVIRT. To do this we have to show for each pair (s1,s2) in the cartesian product SACT X SVIRT, that

$$pc(s1)\ \&\ pc(s2) \Rightarrow sv(s1) \sqsubseteq sv(s2) \qquad (*)$$

where pc and sv refer to the path condition and state vector respectively of a state. First, if pc(s1) & pc(s2) yields a contradiction then (*) is immediately proved. (This corresponds to a pair of paths such that for no input could both paths have been followed.) Secondly, if instruction pointer(s1) is not the same as instruction pointer(s2) there is an error somewhere, either in the program or the inclusion statement so go on to the next inclusion statement or stop. (The actual and virtual programs for the same input must terminate at the same point.) Otherwise, if the two programs have terminated at the same point, we must show that $sv(s1) \sqsubseteq sv(s2)$, that is, for each identifier i in L, $sv(s1)(i) \sqsubseteq sv(s2)(i)$, under the assumption pc(s1) & pc(s2).

There remains the problem of showing that for two expressions a,b, and a logical expression h, that $h \Rightarrow a \sqsubseteq b$. This is done by the use of the following three inference rules:

1) $h \Rightarrow a \sqsubseteq b \leftarrow h \Rightarrow a=b$

2) $h \Rightarrow f((EPS\ x)P(x)) \sqsubseteq b \leftarrow (FA\ x)(h\ \&\ P(x) \Rightarrow f(x) \sqsubseteq b)$

where x is not free in h or b.

3) h => a ⊑ f((EPS x)Q(x)) <- h => (EX x)(Q(x)& a ⊑ f(x))
where x is not free in h or a.

(These rules are read, e.g., to prove h => a ⊑ b, it suffices to prove h => a=b.)

The first of these rules is used when the expressions a and b do not contain any epsilon expressions; the last two serve to reduce the number of epsilon expressions in the formula. Because the last rule requires instantiating an existentially quantified variable, it is usually preferable to use the following particular instance of the rule:

4) h => a ⊑ f((EPS x)Q(x)) <- h => Q(finv(a)) ,

In this rule finv is the inverse of f, and x has been instantiated to finv(a). However, since finv (which may be a relation) could be difficult to find and may not be defined on a, the rule cannot always be applied. If a does not contain an epsilon expression and the rule is applicable then it is actually equivalent to rule 3. Rules 2), 3) and 4) are best understood by interpreting (EPS x)Q(x) as {x: Q(x)} when ⊑ becomes set inclusion. The details of this interpretation are omitted here.

Each of the rules 2), 3) and 4) has a natural generalization to the case when EPS returns a tuple of terms. For example, assuming that $\underline{a}$ and $\underline{x}$ are tuples and that f is a function from tuples to tuples, rule 4) becomes

4g) h => $\underline{a}$ ⊑ f((EPS $\underline{x}$)(Q($\underline{x}$)) <- h => Q(finv($\underline{a}$))

Whenever such tuples have been produced by EPS, inclusion must be shown using these generalized rules and not by individual components as described earlier (which would not usually work anyway).

Finally, if other methods of showing that an array a with bounds lb and ub is included in an array b with the same bounds fail, the following rule is used:

5) a $\sqsubseteq$ b <= (FA i)(lb=<i=<ub => a[i] $\sqsubseteq$ b[i]) .

That is, to show one array is less than another, show that all the corresponding elements are.

2.4.2 Complete programs

To verify a complete program it is simply necessary to verify each of individual inclusion statements assuming the truth of all of the others. This requires the following slight modification in verifying an individual statement. If while symbolically executing the actual program the start of another inclusion statement is reached, then the program enters "hypothetical" mode and starts to execute the virtual program of this new statement. When one of its end points are reached, the program leaves "hypothetical" mode, and continues its symbolic execution. This process is described more formally in Figure 2.3.

```
Verify-specified-program(INCL-STATS)
  foreach INCL-STAT in INCL-STATS do
  Verify-incl-stat(INCL-STAT)


Verify-incl-stat(INCL-STAT)
  with INCL-STAT do
    STATE := mkS(general-symbolic-state-vector,  SP,
                 arbitrary-control-stack,  PREC);
    CUR-INCL-STAT := INCL-STAT;
    SACT := {};
    Ex-act(STATE,INCL-STAT,false);
    SVIRT := {};
    Ex-virt(STATE,INCL-STAT,false);
    Compare(SACT,SVIRT) )


Ex-act(STATE, INCL-STAT, FROM-VIRT)
with STATE do
with INCL-STATE do
repeat (
  if IP is in EP and not(FROM-VIRT) then
     (add STATE to SACT; return);
  FROM-VIRT := false;
  if IP is at sp(IS1) for some IS1 in INCL-STATS
     (other than initially)  then
     (Check that PC => SV(prec(IS1));
      Ex-virt(STATE,IS1,true);  return)
  elseif IP is an assignment (LS := RS) then
     (SV := SV[sym-val(RS)/LS]); advance IP)
  elseif IP is a conditional (if P then s1 else s2) then
     if PC => P then IP := s1
     elseif PC => NOT P then IP := s2
     else (Ex-act(mkS(SV,s1,CS,PC & P),INCL-STAT,false)
           Ex-act(mkS(SV,s2,CS,PC&NOT P),INCL-STAT,false)
           return);
  else (execute instruction normally; advance IP) )
```

Figure 2.3 - Verifying a complete program

```
Ex-virt(STATE,INCL-STAT,HYPMODE) =
with STATE do
with INCL-STAT do
repeat (
  if IP (=ei) is in EP and HYPMODE and the ei is "recursive"
     then (Ex-act(STATE,CUR-INCL-STAT,true); return)
  elseif IP is in EP and HYPMODE  then
     (add STATE to SACT; return)
  elseif IP is in EP then (add STATE to SVIRT; return)
  elseif IP is an assignment (LS := RS) then
     (SV := SV[sym-val(RS)/LS]; advance IP)
  elseif IP is a conditional (if P then s1 else s2) then
     if PC => P then IP := s1
     elseif PC => NOT P then IP := s2
     else (Ex-virt(mkS(SV,s1,CS,PC&P),INCL-STAT,false)
           Ex-virt(mkS(SV,s2,CS,PC&NOT P),INCL-STAT,false);
           return);
  else (execute instruction normally; advance IP) )
```

Figure 2.3 (continued)

2.5 Some exemplary programs

In this section we shall apply continuation induction to several simple programs. The programs are chosen to illustrate how the method copes with various language features and programming techniques, rather than for their own intrinsic interest. Some applications of the method to more complex programs will be discussed later when describing the interactive program verifier and its use.

2.5.1 Recursion

f(X) = if X>100 then X-10 else f(f(X+11))

This program, called the 91-function, is discussed in Manna and Pnueli (1970). There is just one inclusion statement needed to describe it.

Actual program:	body of f
Virtual program:	if X>100 then X-10 else 91
Preconditions:	true
Start point:	entry to f
End points:	exit from f (recursive)
Variables:	none

In this example, as in others, we identify the body of the function with the program consisting of the function's definition and a single call of the function with its formal parameters as arguments. Strictly speaking, there should be another (identical) virtual program corresponding to the call of the function, but the proof of this extra inclusion statement is always completely trivial. That X is an integer is actually a precondition, but we ignore such type

restrictions throughout this work (though we recognize their importance in programming languages and program verification generally). There are no explicit variables to be tested as we are not interested in the value of X at the end of the program, but only in the result of the function call which is always tested.

Figure 2.4 shows the tree of computation paths generated by symbolically executing the actual and virtual programs. Except for the top-level call in the actual program, whenever f is called the virtual program is executed instead. Note that one branch, 91>100, has been cut off at the symbolic execution stage.

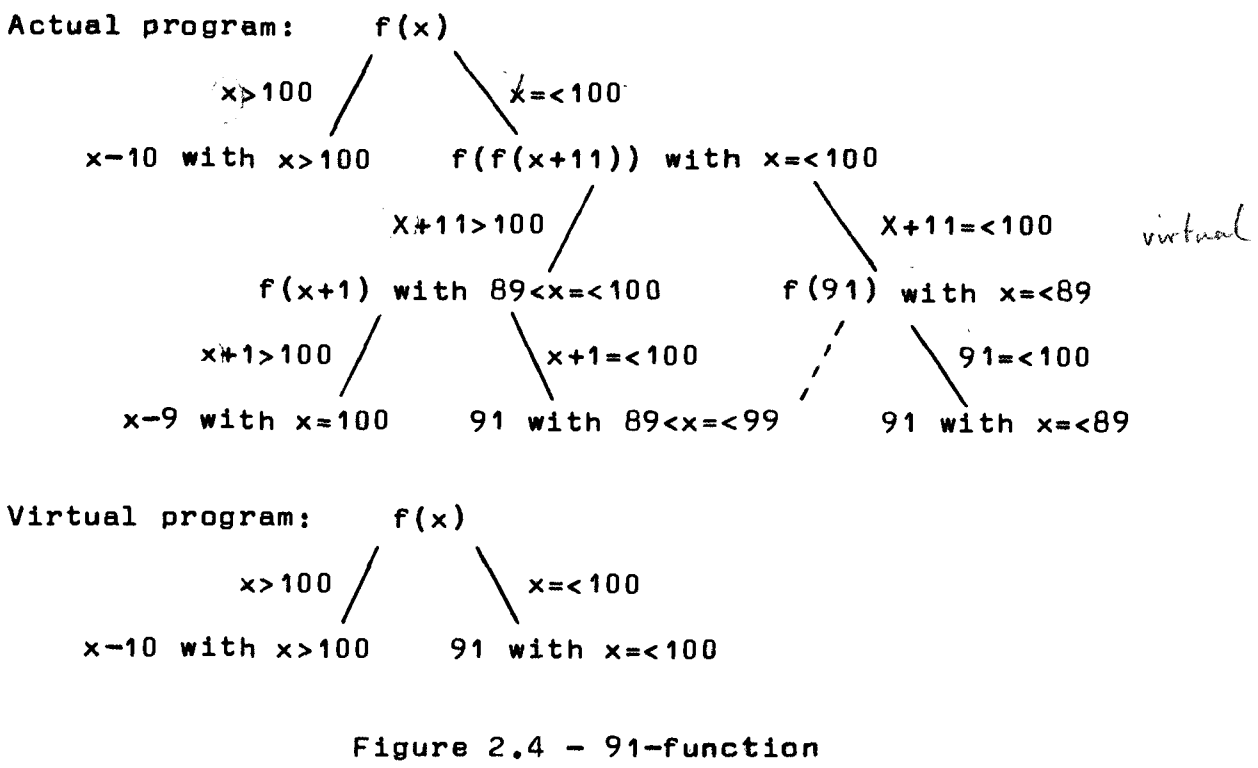


Figure 2.4 - 91-function

Now, by considering all pairs in the cartesian product of the sets of states at the tips of the trees, the verification conditions shown in Figure 2.5 are generated. Notice how the problem has been broken down into simple cases automatically by the theorem generation process. We shall later show how our verifier generates and proves these theorems.

```
x>100 & x>100   =>  x-10 = x-10
x>100 & x=<100  =>  x-10 = 91
x=100 & x>100   =>  x-9 = 91
x=100 & x=<100  =>  x-9 = 91
89<x=<99 & x>100  =>  91 = x-10
89<x=<99 & x=<100  =>  91 = 91
x=<89 & x>100   =>  91 = x-10
x=<89 & x=<100  =>  91 = 91
```

Figure 2.5 - Verification conditions

2.5.2 Iteration and recursion

```
f(N) = (vars R;
        Start:  R := 1;
        Loop:   while N>0 do (N:=N-1; R:= R+N*f(N) );
        Finish: return(R) )
```

This program taken from King (1969) uses both recursion and iteration to compute - guess what? - the factorial function. The declaration "vars R;" declares R to be a local variable of the function. The two inclusion statements needed to verify this program are:

1) Actual program: body of f
   Virtual program: N !
   Preconditions: N>=0
   Start point: entry of f
   End points: exit from f (recursive)
   Variables: none

2) Actual program: while N>0 do (N:=N-1; R:= R+N*f(N))
   Virtual program: R:= R + N! - 1
   Preconditions: N>=0
   Start point: Loop
   End points: Finish (non-recursive)
   Variables: N

Given these statements, the actual proof is straightforward and will not be shown. It depends on the fact that

$$1 + 1*1! + 2*2! + \ldots + (n-1)*(n-1)! = n\ !$$

2.5.3 Non-local jumps

```
S(A) <=
   (Sin: if istip(A) and X=A then (R:=true; goto Finish)
         else if isnode(A) then (S(l A);S(r A));
    Sout: );

Start:  R:=false;
        S(A);
Finish:
```

This program searches a tree A, returning true if one of its tips is equal to X, and false otherwise. (We assume a tree is either a tip or a node which has a left which is a tree and a right which is a tree.) The program uses a recursive function S which does a non-local jump if it finds a tip equal to X. In a more functional language this jump would be written as a call of an escape function. In giving the specifications for this program we assume the existence of a function fr (for fringe) defined by

istip(a) => fr(a) = list(a)

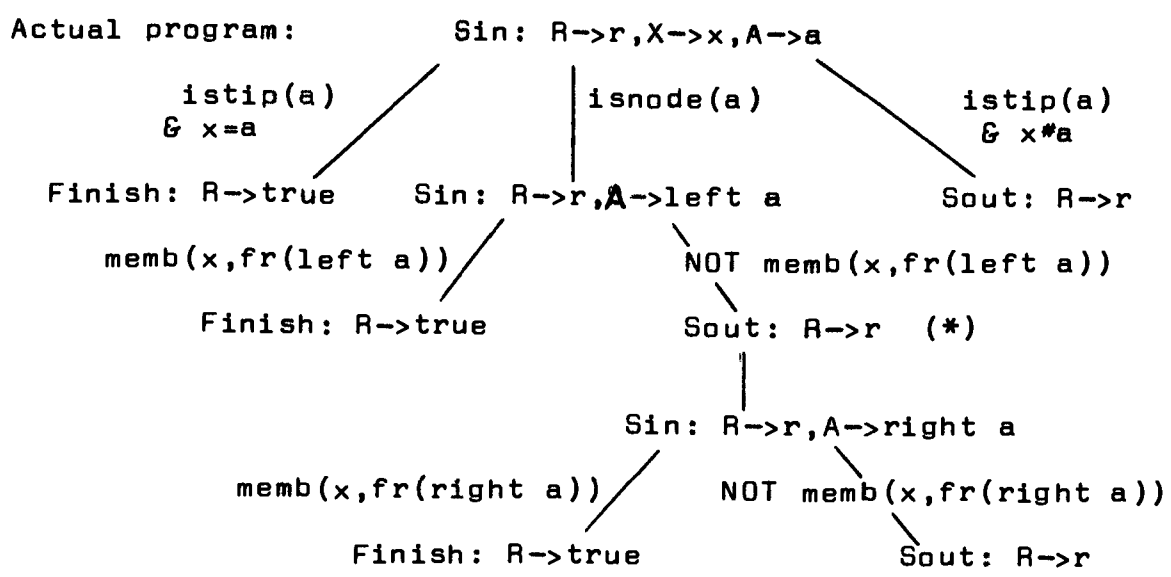
isnode(a) => fr(a) = concat(fr(l a), fr(r a))

and a function memb (for member) which has its normal definition. Then we can express the correctness of the program using the following two inclusion statements:

1) Actual program: whole program
   Virtual program: R:= memb(X,fr(A))
   Preconditions: none
   Start point: Start
   End points: Finish (non-recursive)
   Variables: R

2) Actual program: body of S
   Virtual program: if memb(X,fr(A))
   then (R:=true; goto Finish)
   Preconditions: none
   Start point: entry to S (Sin)
   End points: exit from S (Sout, recursive)
   Finish (non-recursive)
   Variables: R

Again we only consider the second statement since the proof of the first one is trivial. The trees of computation paths and the resulting verification conditions are shown in Figure 2.6. Notice that the computation continues at (*) since Sout is a "recursive" end point. The verification conditions are all easily proved using the definitions of memb, fr and concat, mostly by showing the hypotheses are contradictory.



Verification conditions:

```
istip(a) & x=a & memb(x,fr(a))  => true=true
istip(a) & x=a & NOT memb(x,fr(a)) => true=r
isnode(a) & memb(x,fr(left a)) & memb(x,fr(a)) => true=true
isnode(a) & memb(x,fr(left a)) & NOT memb(x,fr(a)) => true=r
isnode(a) & NOT memb(x,fr(left a)) & memb(x,fr(right a))
    & memb(x,fr(a))  =>  true=true
isnode(a) & NOT memb(x,fr(left a)) & memb(x,fr(right a))
    & NOT memb(x,fr(a))  =>  true=r
isnode(a) & NOT memb(x,fr(left a)) & NOT memb(x,fr(right a))
    & memb(x,fr(a))  =>  r=true
isnode(a) & NOT memb(x,fr(left a)) & NOT memb(x,fr(right a))
    & NOT memb(x,fr(a))  =>  r = r
istip(a) & x#a & memb(x,fr(a))  => r=true
istip(a) & x#a & NOT memb(x,fr(a))  => r=r
```

Figure 2.6 – Non-local jumps

2.5.4 Functional arguments

```
while not(null(L)) do (X:= f(x, hd L); L:= tl L);
return(X);
```

For any function f which is both associative and commutative, this program is included in "lit(L,X,f)", where lit is defined by

null(l) => lit(l,x,f) = x

not(null(l)) => lit(l,x,f) = f(hd l, lit(tl l,x,f)) .

Applying our proof method to the above program, the only non-trivial verification condition resulting is

not(null(l)) => lit(tl l, f(x, hd l, f) = lit(l,x,f),

that is, applying the definition of lit,

not(null(l)) => lit(tl l,f(x,hd l),f)
= f(hd l,lit(tl l,x,f) .

This theorem can now be generalized to

lit(l,f(x,y),f) = f(y,lit(l,x,f))

which is easily proved using associativity and commutativity by structural induction on l. In fact the same original verification condition is generated if structural induction is used from the start. Thus, this example demonstrates that functional arguments can be used in certain cases with our method.

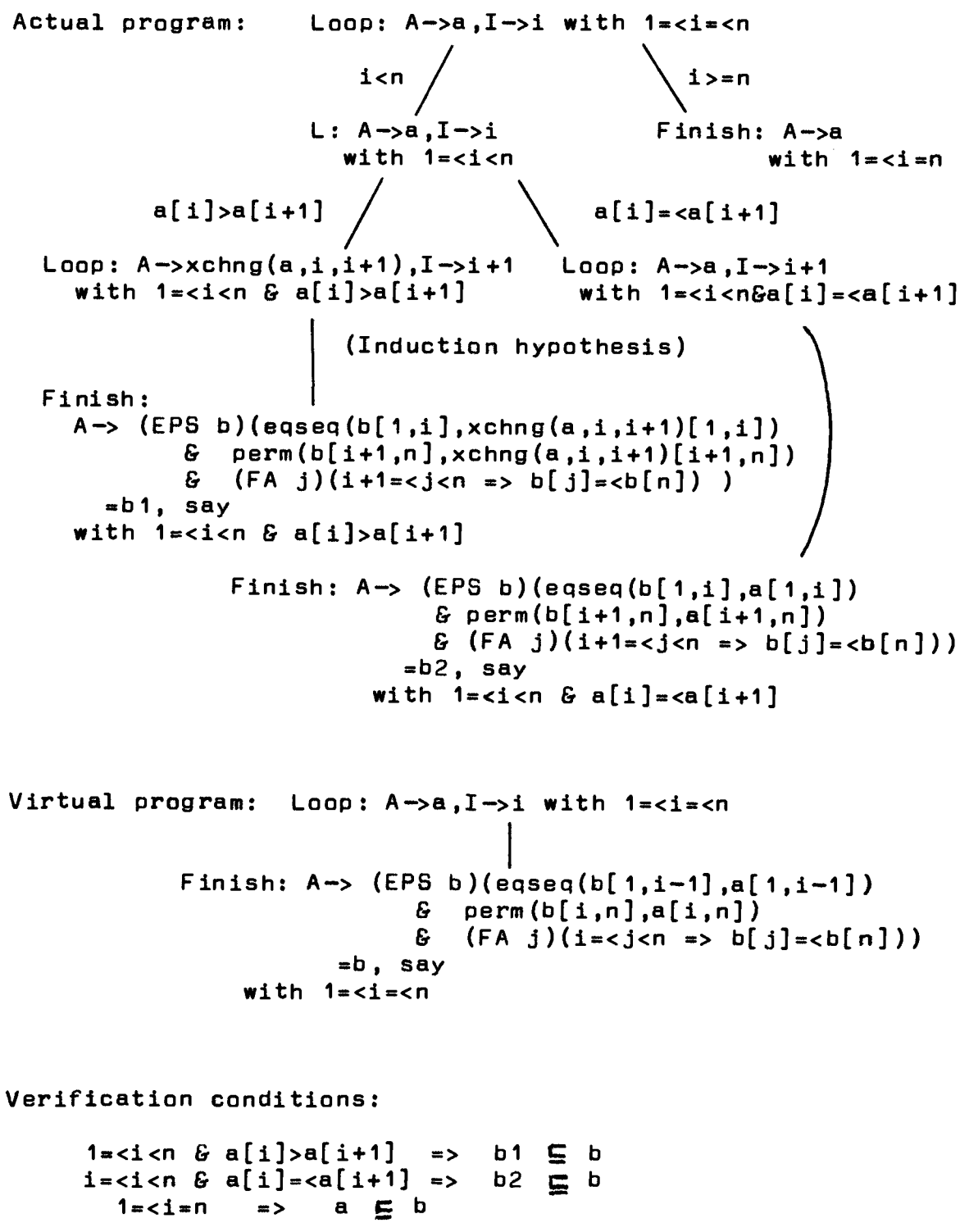
2.5.5 Non-determinism

```
Start:  I:= 1;
Loop:   while I<N do
        L: (if A[I] > A[I+1] then
               (X:=A[I]; A[I]:=A[I+1]; A[I+1]:=X);
            I:= I+1 );
Finish:
```

This is a program which finds the largest element of an array A[1:N] and moves it to the top of the array; it could form the inner loop of a sorting program. Assuming the existence of functions eqseq and perm which test whether one array is equal to, or a permutation of, another, we can write one of the inclusion statements as follows (the other one is similiar):

```
Actual program:  above program from Loop to Finish
Virtual program: A := (EPS B)(eqseq(B[1,I-1],A[1,I-1])
                        &  perm(B[I,N],A[I,N])
                        &  (FA J)(I=<J<N => B[J]=<B[N]) )
Preconditions:   1=<I=<N
Start point:     Loop
End points:      Finish (non-recursive)
Variables:       A
```

Note that the virtual program is non-deterministic. In giving the proof of this inclusion statement we use an operation xchng(a,i,j) which constructs a new array by exchanging the ith and jth elements of the array a. The computation trees and verification conditions for this inclusion statement are shown in Figure 2.7. The values of N are omitted since they never change.



Verification conditions:

$$1 \le i < n \;\&\; a[i] > a[i+1] \Rightarrow b1 \sqsubseteq b$$
$$i \le i < n \;\&\; a[i] \le a[i+1] \Rightarrow b2 \sqsubseteq b$$
$$1 \le i \le n \Rightarrow a \sqsubseteq b$$

Figure 2.7 - Non-determinism

In this example, since it matters, we have been more careful about stating inclusion rather than equality, and will give the proof of a verification condition in more detail than previously. We will use the inference rules given in Section 2.4.1 (previously we have been using the first of these implicitly). Consider the first verification condition. Applying rule 2 with the identity function for f reduces it to

```
1=<i<n & a[i]>a[i+1]
  &  eqseq(b1[1,i],xchng(a,i,i+1)[1,i])
  &  perm(b1[i+1,n],xchng(a,i,i+1)[i+1,n])
  &  (FA j)(i+1=<j<n => b1[j]=<b1[n])
=>
   b1 ⊑ b
```

Applying rule 4 next, again with the identity function for f gives

```
1=<i<n & a[i]>a[i+1]
   & eqseq(b1[1,i],xchng(a,i,i+1)[1,i])      (i)
   & perm(b1[i+1,n],xchng(a,i,i+1)[i+1,n])   (ii)
   & (FA j)(i+1=<j<n => b1[j]=<b1[n])        (iii)
 =>
   eqseq(b1[1,i-1],a[1,i-1])                 (iv)
   & perm(b1[i,n],a[i,n])                    (v)
   & (FA j)(i=<j<n => b1[j]=<b1[n])          (vi)
```

Since xchng(a,i,i+1)[1,i-1] = a[1,i-1], (iv) follows immediately from (i). From (i) and (ii), b1[i]=a[i+1], so perm(b1[i,n],xchng(a,i,i+1)[i,n]) and (v) then follows. To prove (vi) it is only necessary to show additionally that b1[i]=<b1[n]. But b1[i] = a[i+1] < a[i] = xchng(a,i,i+1)[i+1] = b1[j] for some j with i+1=<j=<n. Hence, by (iii), b1[i]=<b1[n]. The proofs of the other two conditions are similiar but easier. To complete the proof of the program, we should also use the definitions of perm and eqseq to prove the facts about them which we have used.

The proof-checker described in Chapters 4 and 5 can be used to do this, and examples of such proofs will be given there.

2.6 Justification of the method

In Burstall (1975) a proof is given that the proof method we have just described is sound, that is, that its successful application to a program does in fact imply that the program is correct. Burstall proves soundness by considering the relation computed from one point in a flow diagram to another and hence shows that the actual program may also be non-deterministic (in the random and not the "backtracking" sense). However the details of his proof are complicated.

In this section we shall outline an alternative method of proving soundness. This method involves translating all programs into systems of recursive equations and then applying computation induction. It is not clear that the resulting proof would be any simpler than Burstall's if all the details were filled in.

2.6.1 Translation to recursive equations

McCarthy (1960) first described how to translate an arbitrary flow-chart program into a set of mutually recursive functions. While this method is well known, it is inadequate for our purposes. The reason is that in McCarthy's method the recursive functions introduced

describe the program from a given point until the end, whereas we want to describe the program from one internal point to another. The motivation for this is that we want to be able to describe the effect of an inner loop without considering the rest of the program containing it. Moreover, McCarthy's method does not allow one to handle escapes when applied to flow-chart programs containing recursive calls. As an example, consider the program:

```
while P(X) do
  (while Q(X) do X:= s2(X);
   X:= s1(X));
return(X);
```

McCarthy's method would translate this into something like:

```
f(X) <= if P(X) then g(X) else X
g(X) <= if Q(X) then g(s2(X)) else f(s1(x))
```

whereas our method would give:

```
f(X) <= if P(X) then f(s1(g(X))) else X
g(X) <= if Q(X) then g(s2(X)) else X
```

Having decided to nest recursive functions in this way, the problem of translating arbitrary flow-chart programs becomes slightly more difficult, but since the solution also deals with escapes from recursive functions, the increased complexity is well justified. To appreciate the difficulty, consider the following program schema (which could correspond to a naive matcher or prime finder):

```
A: while P(X) do
     (X:=a(X);
B:    while Q(X) do
        (if R(X) then (X:=b(X); goto Next);
         X:=c(X));
      goto Finish;
Next: X:=d(X));
Finish:
```

The natural way to write inclusion statements for this program is to have an outer one from A to Finish and an inner one from B to Next or Finish. It is possible to translate this program into recursive functions such that each function corresponds to one of the actual programs by using escapes, but we choose to do it using the more general device of continuations described in Reynolds (1972). A continuation is a function which is added to a given function Fold as an additional argument, giving a new function Fnew, which evaluates Fold and then applies the continuation to this result. That is, for any Fold we can define Fnew by

$$Fnew(x1,...,xn,c) = c(Fold(x1,...,xn)) .$$

Using continuations, we can now write the above program in the following way:

```
A(X,F) <= if P(X) then B(a(X),(LAMBDA u.A(d(u),F)),F)
          else F(X)

B(X,N,F) <= if Q(X) then
                 if R(X) then N(b(X)) else B(c(X),N,F)
            else F(X)
```

where N and F are the continuations, and can be thought of as describing the computations which continue from the labels Next and Finish.

Thus the idea is to translate a specified program by associating a recursive function with the start point of each inclusion statement. In addition to its normal arguments, this recursive function has one extra argument - a continuation - for each end point of the inclusion statement. The details of this translation process depend on the particular language being used and are omitted here.

2.6.2 Validity proof

The argument that the method is sound now goes as follows. A specified program consists of a set of n inclusion statements, say. The whole program can be rewritten as indicated above as a set of mutually recursive functions

$$fi <= Ti[f1,...,fn], \text{ for } i=1,...,n .$$

where each Ti is a monotonic functional, and where each fi is associated with a distinct inclusion statement. That this transformation preserves the meaning of the program is the main gap in our proof. Let gi be the virtual program of that inclusion statement. The proof procedure itself, if successful, has shown that

$$Ti[g1,...,gn] \sqsubseteq gi, \text{ for } i=1,...,n .$$

This is true even though the proof procedure stops immediately before executing the continuations. In the terminology above, it shows that

$$fold(x1,...,xn) \sqsubseteq gold(x1,...,xn) .$$

Since we check that fold and gold terminate at the same point, before executing the same continuation c, and since c

itself is monotonic, we have

$$\begin{aligned} f(x1,\ldots,xn,c) &= c(fold(x1,\ldots,xn)) \\ &\sqsubseteq c(gold(x1,\ldots,xn)) \\ &= g(x1,\ldots,xn,c) \ . \end{aligned}$$

We now have to prove that

$$fi \sqsubseteq gi, \qquad \text{for } i=1,\ldots,n \ .$$

But it is a well-known theorem (e.g. Park 1969) that for a monotonic functional T,

$$T[G] \sqsubseteq G \ \Rightarrow \mu F.T[F] \sqsubseteq G$$

where $\mu F.T[F]$ is the least fixed point of T. Applying this, with (g1,g2,...,gn) in the cartesian product domain for G, yields the result immediately.

Chapter 3. THE INDUCTIVE PROOF METHOD: DISCUSSION

The first part of this chapter discusses an extension and an application of the proof method just described, while the second part compares it in some detail with several other commonly used methods.

3.1 Termination proofs

We start by describing how to extend continuation induction to yield proofs of termination, and hence strong equivalence, of programs. A new component, a decremand (a quantity which is decremented, abbreviated dec), is added to each inclusion statement which corresponds to the body of a recursive procedure or to a cycle, i.e. wherever repetition is possible. (It could in fact be added to every inclusion statement.) This component must contain an expression involving the program identifiers which, for different state-vectors, takes values in some well-founded set, that is, a partially-ordered set with no infinite descending chains. Examples of such well-founded sets are the natural numbers; strings, where a < b if a is a proper substring of b; and lists (as in pure LISP), where a < b if a is a sublist of b, i.e. if a is the hd of b or the tl of b or a sublist of the hd or tl of b. (We use "hd" and "tl" for the LISP "car" and "cdr".)

The proof procedure is now just as before, but with the following addition. Assume we are about to verify an inclusion statement containing a decremand. In the initial symbolic state the symbolic value of the decremand is found and saved. If the program's execution returns to this point, as well as checking the preconditions, the decremand is tested to see that it is now strictly less than the saved value. This will require symbolic reasoning (i.e. theorem proving) of course, rather than simple numerical comparison, for example. If the decremand is indeed less upon return, and this is true for each inclusion statement, then the program must terminate as there are no infinite descending chains. With this extension the proof method now yields strong equivalence if successful, as both programs always terminate. Figure 3.1 below shows the functions Verify-incl-stat and Ex-act of Figure 2.4, modified to incorporate this test.

As a simple example, consider the "counting-up" factorial program of Section 2.5. The decremand of the inclusion statement at Loop is N-I, and the partial order is defined by: x is less than y if 0=<x<y. Since I is increased by one each time the program reaches Loop, and I<N implies 0=<N-(I+1)<N-I, the program (or at least the loop) must terminate.

```
Verify-incl-stat(INCL-STAT) =
  with INCL-STAT do
    STATE := mkS(general-symbolic-state-vector,  SP,
                 arbitrary-control-stack,  PREC);
    CUR-INCL-STAT := INCL-STAT;
    CUR-DEC := SV(DEC);
    SACT := {};
    Ex-act(STATE,INCL-STAT,false);
    SVIRT := {};
    Ex-virt(STATE,INCL-STAT,false);
    Compare(SACT,SVIRT) )


Ex-act(STATE, INCL-STAT, FROM-VIRT) =
  with STATE do
  with INCL-STATE do
  repeat (
    if IP is in EP and not(FROM-VIRT) then
       (add STATE to SACT; return);
    FROM-VIRT := false;
    if IP is at sp(CUR-INCL-STAT)
       (other than initially) then
       ensure SV(dec(CUR-INCL-STAT)) is less than CUR-DEC;
    if IP is at sp(IS1) for some IS1 in INCL-STATS
       (other than initially) then
       (Check that PC => SV(prec(IS1));
        Ex-virt(STATE,IS1,true);  return)
    elseif IP is an assignment (LS := RS) then
       (SV := SV[sym-val(RS)/LS]); advance IP)
    elseif IP is a conditional (if P then s1 else s2) then
       if PC => P then IP := s1
       elseif PC => NOT P then IP := s2
       else (Ex-act(mkS(SV,s1,CS,PC&P),INCL-STAT,false)
             Ex-act(mkS(SV,s2,CS,PC&NOT P),INCL-STAT,false)
             return);
    else (execute instruction normally; advance IP) )
```

Figure 3.1 – Termination proofs

A more difficult example is the program which sorts an array of elements by exchanging pairs of adjacent elements which are out of order, and which terminates when it can make a complete pass through the array without finding such a pair. The decremand of the main inclusion statement of this program is Inversions(A,N), and the partial order is the standard one on the natural numbers. "Inversions" is a function which counts the total number of pairs of elements in the array A from 1 to N which are out of order. It requires some knowledge about permutations and careful reasoning to show that Inversions(A,N) is actually reduced each time around the loop. Sites (1974) was unable to prove this program terminated as he did not have this knowledge about Inversions or permutations. Our theorem prover is capable of doing this sort of reasoning, but we have not yet extended the verification system to do termination proofs.

3.2 Equivalence proofs

Clearly, if we are given two programs A and B such that B satisfies the requirements of 2.3.1 for virtual programs, then we can use this method in an attempt to prove A is included in B. In fact this is what we did with the 91-function. If the program B always terminates, and the termination proof method is applied to program A, then we can even prove that A and B are strongly equivalent.

But it is also possible to prove more interesting programs equivalent. The general method is to treat one of the two programs as the virtual program and translate it, if necessary, into a recursive function, proving separately that it always terminates. Then change the function to a symbolic one (which constructs a symbolic term but is not otherwise evaluated), and use a call of it as the virtual program, saving its definition - a set of recursive equations - for later use. The proof method is then applied as before, and the recursive equations are applied only when comparing the resulting state vectors. This gives a simple method of showing the equivalence (or inclusion) of an iterative function with a recursive function.

As a very simple example consider the following two programs for computing the factorial function:-

```
Start:  I:=1
        R:=1
Loop:   while I<N do (I:=I+1; R:=R*I)
Finish: return(R)
```

and

```
f(N) <= if N=0 then 1 else N*f(N-1)
f(N)
```

In this case we would construct a new symbolic function f1 such that the value of f1(3), say, was simply the term "f1(3)" and save the formulae

n=0 => f1(n)=1,

n≠0 => f1(n)=n*f1(n-1)

for later use. We would then treat the whole first program as an actual program with virtual program f1(N), and treat

the while-statement from loop to Finish as a second actual program with virtual program:

```
Loop:   R:= R*f1(N)/f1(I);
        I:=N;
Finish:
```

The remainder of the proof would then go through as in Section 2.1 above.

As a more interesting example, consider the following program:

```
Start:  C:=0;
Loop:   while N=<100 do (C:=C+1; N:=N+11);
        N:=N-10;
L:      if C>0 then (C:=C-1; goto Loop);
Finish: return(N);
```

We shall prove it is included in the following familiar function:

f(N) <= if N>100 then N-10 else f(f(N+11))

As before we change f to the symbolic function f1 with defining equations

x>100 => f1(x)=x-10,

x=<100 => f1(x)=f1(f1(x+11)),

and construct the inclusion statement which associates the whole actual program with the virtual program f1(N). We now define an auxiliary function Appr (apply f1 repeatedly) with the following defining equations:

y=0 => Appr(x,y)=f1(x),

y>0 => Appr(x,y)=Appr(f1(x),y-1).

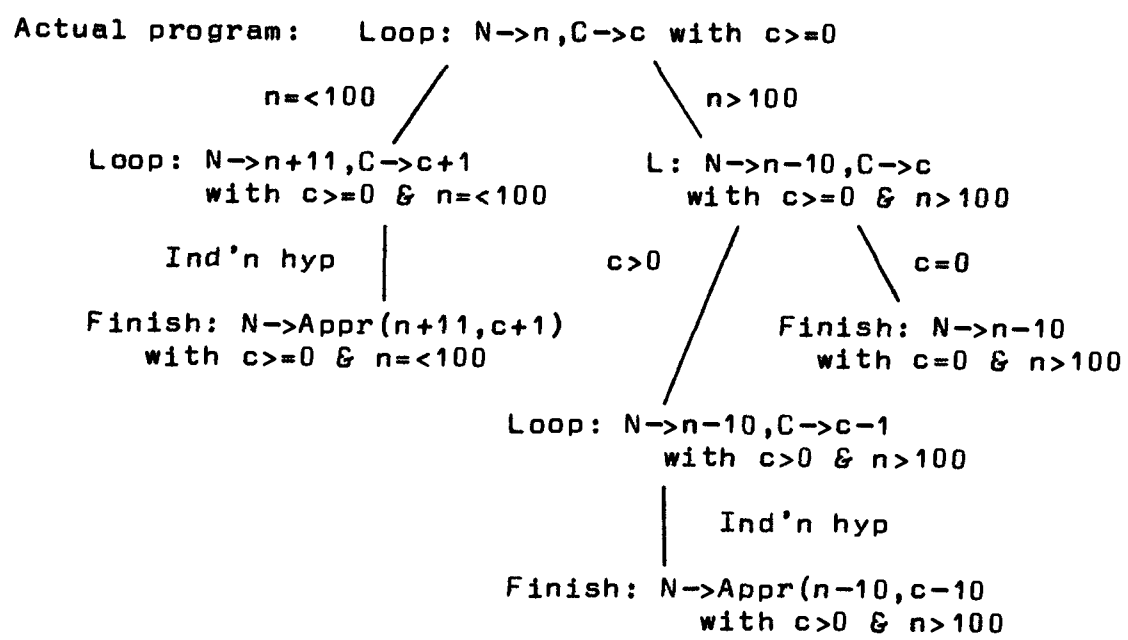
Unfortunately we know of no way to derive this function mechanically. We now construct a second inclusion statement

which starts at Loop, ends at Finish, has the corresponding virtual program:

```
Loop:   N:=Appr(N,C);
Finish:
```

and which has the preconditions c>=0. We concentrate on the proof of this inclusion statement, since the proof of the first one is trivial. In fact even this proof is quite easy. The trees of computation paths of the two programs are shown in Figure 3.2, as are the verification conditions generated. All these verification conditions are easily proved using the definitions of f1 and Appr.

We realise that it will not usually be so easy to find intermediate virtual programs, and that the functions such as Appr and division which need to be introduced may become excessively complex. Nevertheless we believe this is a method which may occasionally be useful.



Verification conditions:

$$c \geq 0 \;\&\; n \leq 100 \Rightarrow Appr(n+11,c+1) = Appr(n,c)$$
$$c > 0 \;\&\; n > 100 \Rightarrow Appr(n-10,c-1) = Appr(n,c)$$
$$c = 0 \;\&\; n > 100 \Rightarrow n-10 = Appr(n,c)$$

Figure 3.2 - 91-function (iterative)

3.3 Comparison with inductive assertions

3.3.1 Flow-diagrams

We assume the reader is familiar with this method of verifying programs. Accounts of it are given by Floyd (1967), Hoare (1969), Elspas et al (1972) and others. Applicable to flow-chart programs, it is closely related to continuation induction. In each case one has to provide, in addition to the overall specifications of a program, some sort of generalized statement at loops: in one case an assertion, in the other an inclusion statement. Both methods do induction on the length of the computation, but whereas with inductive assertions it is on the length from the beginning of the computation to the middle, in ours it is on the length from the middle to the end.

However, there is a much closer connection than this. Continuation induction is really a generalization of the inductive assertion method, and every proof by inductive assertions can be mechanically translated into a proof by continuation induction. We shall demonstrate how this is done by means of two typical examples.

```
Loop: {A1(X)}
      while P(X) do X:=F(X);
Out:  {A2(X)}
```

Here, A1(X) is the loop invariant and A2(X) is the output assertion. Using virtual programs, these specifications can be expressed as the following inclusion statement:

Actual program: while P(X) do X:=F(X);
Virtual program: X:≈(EPS Y)A2(Y)
Preconditions: A1(X)
Start point: Loop
End points: Out (non-recursive)
Variables: X

To verify this inclusion statement we must prove the following three theorems:

a) A1(X) & P(X) => A1(F(X)),

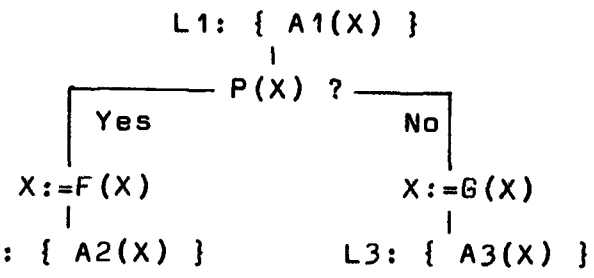
b) A1(X) & P(X) => (EPS Y) A2(Y) ⊑ (EPS Y) A2(Y),

c) A1(X) & NOT P(X) => X ⊑ (EPS Y) A2(Y).

The second of these is always trivially true, and using rule 4) of Section 2.4.1 the third reduces to

d) A1(X) & NOT P(X) => A2(X).

But these two theorems (a and d) are exactly those generated using the inductive assertions directly. If there is a conditional involved the situation is a little more complex. Consider the following flow diagram:



In this case the corresponding inclusion statement is

```
Actual program:    as shown
Virtual program:   X := (EPS Y)(P(Y) => A2(F(Y)) &
                                NOT P(Y) => A3(G(Y)));
                   if P(X) then (X:=F(X); goto L2)
                           else (X:=G(X); goto L3);
Preconds:          A1(X)
Start point:       L1
End points:        L2, L3 (non-recursive)
Variables:         X
```

Verifying this inclusion statement using rule 3 (rule 4 will not work in this case) again reduces the problem to exactly those theorems generated by using the assertions directly. In general, the epsilon expression is written to return a tuple of terms and the generalized forms of rules 2, 3 and 4 used to show the equivalence of the two methods.

Thus we can assume that proofs of program correctness by inductive assertions are simply abbreviations for proofs using virtual programs. There is then a choice between putting the main specifications of the program into the preconditions of an inclusion statement or into the virtual program. Often, less detail is required when the information is put into the preconditions. For example, the loop invariant for the program of Section 2.5.5 is

perm(A[1,N],A0[1,N]) & (FA J)(1=<J<I => A[J]=<A[I])

where A0 is (a ghost variable whose value is) the original array. The corresponding virtual program

```
A := (EPS B)(perm(A[1,N],B[1,N])
          &  (FA J)(I=<J<N => B[J]=<B[N]) )
```

is too weak for the proof to go through as before. The more detailed virtual program

```
A := (EPS B)(eqseq(B[1,I-1],A[1,I-1])
              &  perm(B[I,N],A[I,N])
              &  (FA J)(I=<J<N => B[J]=<B[N]) )
```

is required, though perm could be instead of eqseq. This phenomenon seems to be a weakness of the proof method.

3.3.2 Procedures and Hoare's rules

Because of the above phenomenon it is useful to be able to describe programs using both inductive assertions and virtual programs: the assertions to act as loop invariants, and the virtual programs to describe complete programs and (recursive) subroutines. As an example, the program of Section 2.5.3 is described by:

```
f(N0) <= (vars N,R;
     Start:  N:=N0;
             R:= 1;
     Loop:   assert  R + N! - 1 = N0! ;
             while N>0 do (N:=N-1; R:=R+N*f(N));
     Finish: return(R) )
```

```
Actual program:  body of f
Virtual program: N0!
Preconditions:   N0>=0
Start point:     entry of f
End points:      exit from f (recursive)
Variables:       none
```

Virtual programs can also be used to describe inner loops when the outer loop is described by a loop invariant. The program to invert a permutation, shown in Appendix 4, has its specifications given in this way.

The restrictions on using both inclusion statements and inductive assertions to describe a program are that each cycle in the program must contain either an invariant or the start of an inclusion statement, each subroutine must have a corresponding inclusion statement, and each assertion other

than an output assertion must be followed (dynamically) either by another assertion or by the end of an inclusion statement. To verify such a program we then verify each inclusion statement and each assertion in turn. Verifying an inclusion statement is done as before except that if we encounter an assertion, we check it is true with respect to the current state vector and path condition, replace the state vector by a new, general symbolic one, let the assertion itself be the new path condition, and continue until an end point of the inclusion statement is reached (ignoring any path which reaches an assertion seen before while verifying that inclusion statement). The reason for this treatment of intermediate assertions is that the assertion typically occurs in a loop, so all we know about the state vector is that it satisfies the assertion. To verify an assertion we prove that starting with it as the path condition and symbolically executing the program, every assertion reached is true. We terminate and ignore paths which reach the end of an inclusion statement. If the start of an inclusion statement is reached, the preconditions are checked, the virtual program executed, and the computation continued as before. In this way all the paths through the program will be considered.

In Hoare's theory (Hoare 1969), the statement P{S}Q means that if the assertion P is true before statement S is executed, then the assertion R will be true when (and if) S terminates; axioms and rules for each basic statement S of a programming language effectively define the semantics of

that language, and are used to verify programs written in that language. This approach was extended in Hoare (1971a) to handle recursive procedures and in Clint and Hoare (1972) to deal with jumps out of blocks and with functions. While Hoare's theory is as general as ours, and can be used to define the semantics of a programming language, our method does have one advantage over it, resulting from our use of symbolic execution. This is that we can deal with procedures having side-effects. The virtual program for that procedure simply includes the assignments to the non-local variables. When the procedure is called during another proof these side-effects simply take place as they would during normal execution.

3.4 Comparison with recursion induction

This is a method proposed by McCarthy (1963) for proving the equivalence of recursive functions. He gives the following example of its use. Suppose addition is defined in terms of the operations suc and pre by

$$m+n = \underline{\text{if}}\ n=0\ \underline{\text{then}}\ m\ \underline{\text{else}}\ suc(m)+pre(n)$$

and we wish to prove the theorem

$$suc(m+n) = suc(m)+n\ .$$

Let f(m,n)= <u>if</u> n=0 <u>then</u> suc(m) <u>else</u> f(suc(m),pre(n)); g(m,n)=suc(m+n); and h(m,n)=suc(m)+n. Both g and h can easily be seen to satisfy the defining equation of f, so by recursion induction they are equivalent over the domain on which f is defined, the set of non-negative integers.

McCarthy (1962) extended this principle to apply to flow-chart programs in the following way.

(a) F;

(b) <u>if</u> P <u>then</u> (g; F);

(c) <u>while</u> P <u>do</u> g;

If program (a) can be shown equivalent to program (b), then we can conclude that program (a) is equivalent to program (c) for those state vectors which do not cause program (c) to get stuck in a loop. Saying that program (a) is equivalent to (b) is the same as saying that the program satisfies a functional equation. If another program, G, also satisfies the same functional equation then F is equivalent to G whenever program (c) converges. McCarthy's method of showing the two programs satisfy the same functional equation is by "massaging" them separately until they have the same syntactic structure. For example, to prove

```
A: if   N=0 then goto B;
   R:=N*R; N:=N-1; goto A;
B:
```

equivalent to the program

```
A: R:=N!*R; N:=0;
B:
```

he converts the first program to

```
A: if N=0 then goto B; R:=N*R; N:=N-1;
   A1: if N=0 then goto B1; R:=N*R; N:=N-1; goto A1;
   B1:
B:
```

and the second to

```
A: if N=0 then goto B; R:=N*R; N:=N-1;
   R:=N!*R; N=0;
B:
```

using properties of the factorial function. Thus both programs satisfy a relation of the form:

"program"

is equivalent to

A: if N=0 then goto B; R:=N*R; N:=N-1; "program"; B:

and are hence equivalent whenever the first of them terminates.

This seems an unnecessarily complicated procedure, having to syntactically transform both programs, but in essence it is again the same as ours. However, continuation induction has the following advantages over recursion induction as McCarthy described it.

(1) A minor advantage is that our method makes explicit the way to handle nested loops. McCarthy gave no examples (in the papers we have seen) containing them, and his method of translating flow-chart programs into recursive functions is not suitable for the task, as we observed in Section 2.6.

(2) Proofs seem more natural and easier to find in our method as the "massaging" of the programs is done automatically in the course of the symbolic execution.

(3) Strong equivalence (i.e. termination) can often be proved in the course of the main proof.

(4) Our method is capable of proving properties of programs which contain escapes or, equivalently, multi-exit loops in flow-chart programs. This would seem to be its main advantage over recursion induction. Of course, at the time McCarthy did his work the concept of a continuation was not explicitly available to him as it was to us.

3.5 Other related methods

Another method based on symbolic execution (or hand simulation) is described in Burstall (1974). The basic inductive statement of this method is of the form:

> "Starting from label K with state vector X=x0, Y=y0, ... satisfying P(x0,y0,...), for all i such that 0=<i=<n, the program eventually reaches label L with the state vector X=x(i), Y=y(i), ... satisfying Q(x(i),y(i),...)."

Thus the method automatically yields proofs of total correctness whereas all the other methods we have discussed only yielded partial correctness. The main difference between it and continuation induction is that it does induction on the data whereas continuation induction does induction on the length of the computation.

Burstall and I programmed a simple implementation of the method in 1972 and proved some trivial programs using it. We eventually rejected it in favour of the present method because it needed a more sophisticated logical apparatus to do the inductions. Moreover the continuation proof method seemed easier to implement at the time.

However, for certain types of programs, such as iterative translations of recursive programs which operate on trees, for example, it can be very useful. In Topor (1974), Appendix 6 of this thesis, I gave a proof of the Schorr-Waite list marking algorithm. This is an example where a data induction approach is clearly preferable to doing induction on the length of the computation, and both the inductive assertion method and continuation induction are unsuitable.

Recently, and independently of our own work, Mills (1975) and Basu and Misra (1975) have shown how continuation induction can be used to prove the correctness of iterative programs constructed using while statements alone (loop programs). In each case they assume that the "virtual program" is given initially as some function f and they attempt to show that the function computed by the while statement is equal to f whenever the while statement terminates. They formulate the principle as follows: the equivalence "f = <u>while</u> p <u>do</u> g" holds if and only if for every (x,y) in f the iteration terminates and p(x) => y=f(g(x)) & NOT p(x) => y=x. Basu and Misra also

show that given the function f which the loop computes one can find a loop invariant which suffices to prove the correctness of the program. This provides a dual to our result that proofs using inductive assertions can be translated into proofs by continuation induction. However, when the program has the form "initialization; loop" and the function (intended to be) computed by the whole program is known, our experience indicates that it is often just as difficult to find and describe the function computed by the loop alone as it is to find a loop invariant directly. This can be seen from the examples we have already studied. In neither of these two papers are these methods applied to the other control structures such as arbitrary flow-charts, recursion and escapes which we have considered.

Chapter 4. AN INTERACTIVE PROGRAM VERIFIER

4.1 Overview

Any reasonable program verifier will clearly require human assistance to prove some of the verification conditions. It will also require a certain amount of knowledge about the problem domain on which the program is operating. Our aim has been to develop a system which a programmer could use to verify a moderately complex program by interactively proving the verification conditions and extracting the required facts about the domain in the process. We expect that it might take some time to interactively verify a program in this way, possibly longer than the time which would be taken to "debug" the program, since a period of thought away from the terminal will usually be required.

The system we have implemented verifies POP-2 programs and is written in POP-2. (A brief description of this language is given in Appendix 1.) It is basically an implementation of the continuation induction proof method described in Chapters 2 and 3, extended to allow inductive assertions but not functional arguments. To use it the user provides a POP-2 program together with its specifications as a set of inclusion statements and/or inductive assertions. The specifications can be written in terms of new primitive functions declared by the user. The system then compiles the specified program and attempts to verify each inclusion

statement and assertion in turn. If the system cannot prove a particular verification condition it asks the user for help. The user then has available a wide repertoire of commands he can give in guiding the system to a proof; in this mode the system acts as a proof-checker. In particular, the user can provide or use facts about the new primitive functions in the form of reduction and inference rules. It is also possible to execute the specified program with actual numerical data either to satisfy the sceptic or, having verified the program, to actually use it.

The most complex programs we have verified using the system are two sorting programs and a program to invert a permutation "in place". The permutation example was done after the rest of this work had been completed, and took about ~~about~~ two weeks to find and define the appropriate concepts, give an adequate specification, and then verify the program's correctness. We now describe the different aspects of the verification system in more detail. The reader should refer to the examples in Chapter 6 while reading this description.

4.2 Input languages

The program to be verified must be written in a subset of POP-2 as a normal function which may call other functions. Each non-trivial function thus called must be described by a separate inclusion statement. The subset of

POP-2 used allows integers with their normal operations; truthvalues; one-dimensional arrays of integers; and lists with constructive operations but without destructive updating.

The standard functions and variables allowed are: < > =< >= = /= + - (binary only) * div rem (not //) :: <> atom back cons dest erase false front hd identfn jumpout nil not null tl true undef.

The syntax words allowed are:( ) . , ; :: -> => and close else (but not inside LOOPIF) end function goto if lambda loopif nonmac nonop or then vars.

This is quite a restriction on the language, but there is still infinite scope for writing complex programs. An extension is that epsilon expressions are allowed, having the syntax (ANY <varlist>)<expression> or <ANYARR <varlist>)<expression> (for arrays) where <varlist> is either a word or a list of words. The syntax of the language is also changed in the following way:

1) Every label must be preceded by a dollar sign ($)

2) Subscripted variables must be written e.g. A \ (I+J), and arrays are declared differently (see later).

A virtual program is written as a POP-2 imperative sequence using the same subset of the language, but naturally without loops or recursion. If the inclusion statement to which it belongs has more than one end-label, each exit from the virtual program must have a GOTO

<end-label> at it. Virtual programs may also contain new primitive functions as described in the next section; they may refer to array sequences e.g. A<<I,I+J>>; they may contain assignments to arrays e.g. (ANYARR B)P(B) -> A; and they may contain assignments to array sequences e.g. A<<1,J>> -> B<<I+1,I+J>>. Note also that the verifier assumes that the body of a function terminates immediately before the output locals are put on the stack, so this must be considered when writing virtual programs.

A specified program must then have the following form:

```
<function-definition *>;

<declaration-of-input-variables>;
<actual-parameter-list> -> INITARGS;

<inclusion-statement 1>;
            .
            .
            .
<inclusion-statement n>;
```

The syntax of each inclusion statement is as follows:

```
<incl-statement>   ::= ISTAT <incl-location>
                       VIRT  <virtual-program>
                       UNDER <preconditions>
                       WRT   [<identifier*>]

<incl-location>    ::=  BODY (REC ?) <function-name>  /
                        <begin-label> TO [<end-label-list>]

<end-label-list>   ::=  (<end-label> (REC ?) *)

<virtual-program>  ::=  <imperative-sequence>

<preconditions>    ::=  <logical-expression>
```

Examples of specified programs are given in Appendix 4.

The program to be verified may also contain assertions after labels at key points throughout the program. The syntax of these statements is simply

<assertion> ::= ASSERT <logical-expression>;

Every assertion except for an input assertion must be preceded by a label. An output assertion is identified by having the corresponding label OUT.

4.3 Definitions and rules

In giving the specifications for a program, new primitive functions are usually required to express the program's intention. This is specially important to enable the verifier to deal with programs operating on new domains. The verification system allows the user to declare such functions by typing, for example, DECFUNS F 2 G 3; which declares F to be a symbolic POP-2 function of 2 arguments and G of 3. It also allows him to specify their properties either as simple predicates or as reduction or inference rules. Amongst these properties the definition of the function is singled out; whilst the other properties may be taken as given, for a proof to be complete, they must be shown to follow from the definition. The functions we have used in verifying programs are SEQOF, EQSEQ, ISIN, PERM, ORDERED, MEMB and FRINGE. The function application SEQOF(A,I,J) is also written A<<I,J>>. The definitions of these functions are given in Appendix 2.

The simple properties which a function can be declared to have are associativity, commutativity, transitivity and whether it takes numerical values. These are specified using the doublets ISASSOC, ISCOMM, ISTRANS and ISNUMFN respectively. New boolean predicates like these can be introduced by the user and this facility proved very useful while developing the system. Functions can also be assigned an identity and a zero, using the doublets IDENTOF and ZEROOF.

We next describe the two sorts of rules which can be used. These rules are defined using variables declared by, e.g., DECLVARS X Y; The collection of rules defining and describing a particular function constitutes, in effect, a mini-theory of that function; all the rules relating to this function are loaded with the verifier whenever a program involving it is being verified. A list of all the derived rules (rules or facts which are not definitions) is given in Appendix 3; we have proved some but not all of these using the interactive theorem prover. The description of how these rules are actually used is deferred until the next chapter.

4.3.1 Reduction rules

Reduction rules are used to define functions which are not predicates, and correspond roughly to the antecedent theorems of PLANNER or the demons of QA4. They are rules for rewriting (or transforming) expressions into equivalent but

"simpler" ones. In general, a rule is only applicable to an expression if the subterms of an expression satisfy certain restrictions. These restrictions are written as a list of conjuncts:

```
<conjunct-list> ::= [% <expression-seq ?> %]
<expression-seq> ::= <expression> (,<expression> ?*)
```

Reduction rules are then defined by:

```
<reduction-rule> ::= (WHEN <conjunct-list>, ?)
                        <expression> ==> <expression>
```

For example, integer division which is treated by the algebra system as a user-defined function, has some of its properties given by the following rules:

```
WHEN [% 0 =< X, X<Y%], X DIV Y ==> 0;
WHEN [% X>=Y, Y>0 %], X DIV Y ==> 1+(X-Y)DIV Y;
X DIV 1 ==> X;
(X*Y)DIV(X*Z) ==> Y DIV Z;
WHEN [% X REM Y = 0 %], (X DIV Y)*Y ==> X;
```

The first two of these constitute the definition of division when restricted to the non-negative integers.

4.3.2 Inference rules

Inference rules are used to define and give properties to predicates; they correspond to the consequent theorems of PLANNER or the goal class rules of QA4. They are inference rules in the sense that if their hypotheses are true then their conclusion must also be true, but they are always used in a top-down manner: to prove the conclusion,

try to prove the hypotheses. Inference rules are defined by:

```
<inference-rule> ::= IR <rulename>
                         <expression> <== <conjunct-list>

<rulename> ::= <identifier>.
```

Two typical inference rules are:

```
IR ORD1
   ORDERED(A<<I,J>>) <== [% I>=J %];

IR ORD2
   ORDERED(A<<I,J>>)
   <== [% I<J, A\I<A\(I+1), ORDERED(A<<I+1,J>>) %];
```

Inference rules are given names so they can be referred to when using them interactively. Note that we have explicitly written "if and only if" definitions as two or more separate implications in an equivalent but heuristically more useful way. Thus instead of writing the rule

```
ORDERED(A<<I,J>>)  <=>  I>=J
                         OR A\I =< A\(I+1)
                              & ORDERED(A<<I+1,J>>) ,
```

we write the four equivalent rules

```
ORDERED(A<<I,J>>) <== [% I>=J %]

ORDERED(A<<I,J>>) <== [% I<J, A\I=<A\(I+1),
                          ORDERED(A<<I+1,J>>) %]

A\I=<A\(I+1) <== [%(EX "J")(I<J & ORDERED(A<<I,J>>))%]

ORDERED(A<<I,J>>) <== [% I-1<J, ORDERED(A<<I-1,J>>)%] .
```

Notice also that we have used recursion in writing both the definitions and properties of most new functions. This was done to make them more immediately applicable to the

theorems produced by the verification generator. However definitions using quantifiers can also be used, and in fact we have used both the recursive and non-recursive definitions of ordered to prove the two sorting programs correct. The alternative definition was expressed as the two inference rules:

```
ORDERED(A<<I,J>>)
<== [% (FA U)(I=<U & U<J =>> A\U=<A\(U+1)) %] ,

A\U =< A\(U+1)
<== [% (EX [I J])(I=<U & U<J & ORDERED(A<<I,J>>)) %] .
```

Using this definition the proofs required slightly more user intervention since the system does not automatically invoke the proof by cases which corresponds to the recursive definition. However, because fewer non-recursive rules are required to express the same properties as several recursive rules we are currently modifying the system so that they can be used more easily.

4.4 Verification condition generator

As stated above, the verifier is basically an implementation of continuation induction extended to allow inductive assertions. Since the POP-2 user has access to the stack, this is added to the machine state as an additional component. When the specified program is "compiled", a list of all the inclusion statements is formed. The user can then ask either for all of them or for a particular one to be verified. An attempt is made to

actually prove the verification conditions generated from each inclusion statement before starting on the next one. After verifying the inclusion statements, each assertion present is verified in turn as described in Section 3.3.

To compare two states s1 and s2, found by running an actual program and a virtual program, the verifier has to prove a theorem of the form

$$pc(s1)\ \&\ pc(s2) \Rightarrow sv(s1) \sqsubseteq sv(s2)$$

where pc is the path condition and sv is the state vector. The algorithm it uses to do this is as follows:

1) If for each variable v, sv(s1)(v) is identical to sv(s2)(v), then exit with success.

2) Otherwise, if there exists a conjunct c in pc(s2) such that pc(s1) => NOT c, then exit with success.

3) Otherwise try to prove that for each variable v, $sv(s1)(v) \sqsubseteq sv(s2)(v)$.

In the successful cases the system also checks that the two programs terminated at the same end point and that the stacks are equal. If the proof fails, the system gives an appropriate message, and the user can direct the theorem prover to work interactively on either 2) or 3). However if the theorem is clearly false, an error in the program or its specifications is indicated. The error can be located by observing the path condition of the actual program which caused the error to appear.

It is also possible to make the system generate verification conditions without trying to prove them. More usefully, the user can specify that no proofs should be attempted at conditionals during symbolic execution. Although this may lead to unnecessary branches of the computation tree being generated the resulting theorems are invariably easy to prove; moreover, especially in cases where there are no conditionals in the virtual programs, some processing time will be saved. The user can also choose whether to apply rule 3 or 4 (of Section 2.4.1) when proving inclusion of epsilon expressions; rule 4 is the default case.

Output from the verifier can be sent in varying degrees of detail either to disc or to the terminal. A record of any interactive proofs is always saved on disc, and at the end of verifying a program the user can save any unproved verification conditions on disc for later analysis.

The most significant aspect of the way verification conditions are generated is the use made of the normal POP-2 compiler and run-time system. In fact we have used these exclusively and have written neither our own parser nor our own interpreter. All the control aspects of symbolic execution (statement sequencing, jumps, procedure entries and exits) are handled by the normal run-time system. Thus any correctness results proved are true with respect to the actual implementation of the language, rather than with respect to some abstract definition of it. This has the

curious consequence that even if there are errors in the language implementation (with respect to its definition), programs verified using that implementation will always run correctly on it. Moreover it enables programs using non-standard language features (such as escapes) to be verified without having to give abstract definitions of these features beforehand. The disadvantage is that there is no guarantee that programs verified on one implementation of the language will run correctly on another.

4.5 Implementation

4.5.1 Summary

The system is implemented in POP-2 and runs on a PDP-10 (with a KA-10 CPU). When the system is loaded together with all the inference and reduction rules it occupies about 47K 36-bit words. The breakdown of this into the system components is as follows:

POP-2 system	14 K
Algebra system	12 K
Verification condition generator	5 K
Theorem prover	6 K
Inference and reduction rules	10 K
Total	47 K

Normally when using the system we only compile those rules involving the functions occurring in the particular example

being studied. Note also that a list cell in POP-2 requires 3 PDP-10 words. The system spends most of its time in algebraic simplification and matching.

In the remainder of this section we shall describe the implementation of the verification condition generator in more detail. The key to implementing the proof procedure described in Figure 2.3 by using the normal run time system is the use of recursive coroutines. One coroutine is the actual (or virtual) program being symbolically executed; the other is a recursive function called MONITOR which actually controls the symbolic execution. Control and information is passed between these two coroutines by a function SWAP of one argument and one result - a message - which uses a global saved state called STATE for resuming a coroutine. Since there are only the two coroutines neither this saved state nor, equivalently, the name of the coroutine need to be given as an argument to SWAP.

4.5.2 MONITOR

MONITOR is the function which actually simulates the functions Ex-act and Ex-virt of Figure 2.3. It is resumed whenever the actual (or virtual) program reaches a label or a conditional; the other cases are handled by the normal run-time system. A slightly simplified, descriptive definition of MONITOR is shown in Figure 4.1. The only component of the state which needs to be given to MONITOR is the path condition as the others are again all held

implicitly by the run-time system. At the start of an inclusion statement calling SWAP with TRUE or FALSE runs the virtual or actual program respectively. HYPMODE is initially FALSE, and CURINCLSTAT is initially the inclusion statement being verified. Notice how closely MONITOR corresponds to the functions in Figure 2.1. The main difference is that it only calls itself recursively at conditionals. The advantage of writing the function recursively is that the stack of branch points is maintained automatically as was suggested by Stansfield (1972).

The extension to handle inductive assertions is done within the same framework. Assertions are stored as expressions and are associated with the labels at which they occur. Some extra cases are added to MONITOR and OOLABEL as a label may now have an assertion. The procedure given in Section 3.3 is then followed in a straightforward fashion.

```
FUNCTION MONITOR MESS PATHCOND HYPMODE => STATES;
VARS RETFLAG;
LOOP:
  IF ATOM(MESS) THEN SWAP(MESS)->MESS
  ELSEIF HD(MESS)="COND" THEN
     DOCOND(HD(TL(MESS)),PATHCOND)->STATES; RETURN
  ELSEIF HD(MESS)="LABEL" THEN
     DOLABEL(HD(TL(MESS)),PATHCOND)->RETFLAG;
     IF RETFLAG="RETURN" THEN
        UNITSET(PATHCOND::TL(MESS))->STATES; RETURN
     ELSE SWAP(RETFLAG)->MESS CLOSE
  ELSE ERROR() CLOSE;
  GOTO LOOP
END;


FUNCTION DOCOND TEST PATHCOND => STATES;
VARS SAVSTATE;
  IF "PATHCOND implies TEST" THEN
     MONITOR(1,PATHCOND,HYPMODE)->STATES; RETURN
  ELSEIF "PATHCOND implies NOT(TEST)" THEN
     MONITOR(0,PATHCOND,HYPMODE)->STATES; RETURN
  CLOSE;
  STATE->SAVSTATE;
  MONITOR(1, TEST & PATHCOND, HYPMODE) -> S1;
  SAVSTATE->STATE;
  MONITOR(0, NOT(TEST) & PATHCOND, HYPMODE) -> S2;
  UNION(S1,S2) -> STATES;
END;


FUNCTION DOLABEL LABEL PATHCOND => RETFLAG;
VARS I;
  IF ISEXIT(LABEL,CURINCLSTAT) THEN
     IF NOT(HYPMODE) THEN "RETURN"->RETFLAG; RETURN CLOSE;
     FALSE->HYPMODE;
     SAVINCLSTAT->CURINCLSTAT;
     IF "LABEL is a recursive endpoint" THEN
        FALSE->RETFLAG; RETURN
     ELSEIF ISEXIT(LABEL,CURINCLSTAT) THEN
        "RETURN"->RETFLAG; RETURN
     CLOSE
  CLOSE;
  IF ISSTART(LABEL) THEN  -> I;
     COMMENT 'LABEL is the start of inclusion statement I';
     CURINCLSTAT->SAVINCLSTAT; I->CURINCLSTAT;
     "check preconditions of CURINCLSTAT";
     TRUE->HYPMODE;
     TRUE->RETFLAG
  ELSE FALSE->RETFLAG CLOSE
END
```

Figure 4.1 - MONITOR

4.5.3 Source program transformations

Compiling a specified program is actually done in two passes. In the first pass variables are declared and a record is made of all the inclusion statements. In the second pass new labels are inserted at the beginning and end of every function body, and the actual program alone is compiled by the POP-2 compiler after the transformations shown in Figure 4.2 are made using macros. These changes are necessary to enable the program to communicate with MONITOR. The conditional statement in parentheses is only inserted if the label is the start label of an inclusion statement.

Before	After

`L: <program>`	`L: SWAP([%"LABEL",L,current-state-vector%])`
	`-> RESULT;`
	`(IF RESULT THEN`
	`<corresponding-virtual-program>`
	`CLOSE;)`
	`<program>`

`IF <expr>` `THEN ...`	`IF SWAP([%"COND",<expr>%])` `THEN ...`

`L: LOOPIF <expr>`
`THEN ...`
`CLOSE`
	`L: IF <expr> THEN`
	`...`
	`GOTO L; CLOSE`

Figure 4.2 - Program transformations

4.6 Discussion

The verification condition generator, at least when restricted to proofs by inductive assertions, is very similiar to Deutsch's and contrasts with those of Igarashi, London and Luckham, and the use of LCF. The difference is that our system uses the operational semantics of the language rather than an axiomatic definition. Using continuation induction enables this approach to be applied to a wider range of programs than it was previously. Moreover, using the acc, chng and xchng functions on arrays and assignments to array variables avoids a proliferation of cases as had been previously observed.

Also, using forward evaluation to construct verification conditions allows the verification conditions to be simplified before they reach the theorem prover. While our system does not do as much simplification as Deutsch's - in particular it does not treat equality tests as assignments - it does simplify all expressions as they are constructed, thereby saving work later, and sometimes cutting off impossible computation paths.

Practically, the use of an actual language system has had both advantages and disadvantages: it saves some work of course, but it has occasionally been awkward conforming to POP-2 syntax and keeping variables distinct.

The verification systems most similiar to ours are (i) the one being developed by Good, London and Bledsoe (1974) and (ii) the one being developed at Stanford (von Henke and Luckham 1974, Suzuki 1974). Both these systems support the use of abstraction in writing specifications and have the ability to easily add new knowledge about the abstract functions being used. The main differences from our system is that they both use an axiomatic definition of their programming language (PASCAL) for generating verification conditions and only use the inductive assertion method. The simplifier and theorem prover of the Stanford system is completely automatic, whereas our system and Good's rely on interactive theorem proving.

There is also a difference in the way the systems are used. In each system unsuccessful proofs are used to indicate which properties of the new (abstract) functions are required. However, whereas we isolate certain basic properties as definitions and (attempt to) prove the other properties from them, the Stanford group ensure that all their properties are consistent by observing that they have a model. It is not clear how Good et al. ensure that their properties are consistent.

Chapter 5. ALGEBRAIC MANIPULATION AND INFERENCE

In this chapter we shall describe the way we represent and manipulate symbolic expressions, how the automatic theorem prover works, what interactive commands are provided, and how they are used.

5.1 Algebraic manipulation

At the heart of the theorem prover and the verification condition generator lies a general purpose algebra system, or more accurately, a symbolic manipulation system. This system is, in many ways, similar to those used by King and Deutsch, however, in other ways, it is rather more general. The system is used to construct and manipulate symbolic expressions in the domain of integers and also in other domains e.g. that of lists. General routines for applying properties of operators are used to put expressions into a canonical form whenever possible. It is easy to add new functions together with brief descriptions of their properties which can then be used to simplify expressions involving the new functions. The system also contains routines for simplifying relational expressions and conjunctions of relational expressions, for applying substitutions, and for matching one expression against another. Each of these aspects is discussed in turn.

We start by giving some definitions. Two expressions, e1 and e2, are said to be identical, written e1==e2, if they are the same string of symbols or, if presented in abstract syntax, they have exactly the same form. That is they are syntactically the same. They are said to be equivalent (or equal), written e1=e2, if for any assignment of values (in the domain) to their (free) variables they have the same value in the domain. That is they are semantically the same. Thus a+b and b+a are equivalent but not identical. An f-canonical form for a class of expressions C is a computable mapping f:C->C such that:

1) for all e in C, f(e)=e, and

2) for all e1,e2 in C such that e1=e2, f(e1)==f(e2).

Often the function f is implicit, and we simply talk about canonical forms: an expression e is said to be in canonical form if f(e)==e. A normal form is a weaker concept applicable to algebraic expressions which maps all expressions equivalent to zero into zero, though the distinction between normal and canonical forms is often ignored. Simplification is a still weaker operation which transforms an expression into an equivalent (simpler) expression which may or not be in normal form.

Canonical forms are important in both algebraic manipulation and theorem proving, since if a canonical form exists, and all expressions are put into it, then two expressions are equivalent if and only if they are identical. Being able to detect equivalence immediately in this way is a big help in both further simplification and

deduction. Normal forms are useful because expressions which are equivalent to zero are always reduced to zero, thereby simplifying subsequent processing. Moses (1971) reviews known results concerning the existence of normal (called zero-equivalence) and canonical forms for various classes of algebraic expressions.

5.1.1 Expressions and simplification

Our system deals with a simple but general class of expressions. The range of values, or domain, of these expressions can vary, depending on the application; they could be truth-values, numbers, lists, arrays or functions. Members of the domain are called specific items (or constants). The expressions have the following abstract syntax:

An expression is either a primitive expression
or a function application
or a quantified expression.

A primitive expression is either a specific item
or a variable.

A function application has a funname which is a word
and a funargs which is a expression-list.

A quantified expression has a bdvar which is a variable
and a quantifier which is FA, EX or EPS
and a body which is an expression.

Internally, specific items represent themselves, variables are represented by words, function applications are represented by lists whose head is the funname and whose tail is the funargs, and quantifiers are represented by a three element list containing the quantifier, bdvar and body

respectively. Associative functions are variadic, so both +(1,2) and +(1,2,3) could be expressions. Every function has two variants: a symbolic function which, for addition, returns +(a,b) as the value of a+b, and the original (specific) function, possibly modified to take a variable number of arguments, which returns 3 as the value of 1+2.

All expressions are automatically simplified as they are constructed. This simplification aids later processing by increasing the number of equivalent expressions which are also identical; for the class of multivariate polynomials over the integers, the resulting simplified form is actually a canonical form. To simplify an expression the following steps are performed in order:

1) Primitive and quantified expressions are left unchanged.

2) Otherwise the expression is a function application. the function's zero is in the argument list it is returned as the result.

3) If all the arguments are specific items, the original function is applied to them and the result returned as the simplified expression.

4) If the function distributes over any other functions the appropriate transformation is applied, recursively simplifying the inner terms. The two functions considered in this category are multiplication (a*(b+c) ==> a*b+a*c), and disjunction ((a&b) v c ==> (avb) & (avc)).

5) If the (possibly new) function is associative, all applications of this function as arguments are brought to the top level. For example, +(a,+(b,c)) simplifies to

+(a,b,c). Note that this cannot undo steps 1) to 4) as all the inner terms have already been simplified.

6) Next, if the function is commutative, its arguments are sorted into lexicographic order (with numbers before words before lists, and words ordered as they were declared) ignoring constant factors of products when simplifying sums and exponents of powers when simplifying products, and combining arguments whenever possible. An example should make this clear: a*c+a^2+b+3*a+2+5*a reduces to 2+8*a+a^2+b+a*c.

7) If the function is associative and not commutative another pass is made through the expression combining terms whenever possible.

8) Finally, any occurences of the identity ae removed, and any expressions of the form f(a) where f is associative are simplified to a.

With the exception of only two functions, a function's symbolic variant merely constructs a new term which is then handed to the simplifier described above. The exceptions are subtraction, which immediately simplifies a-b to a+(-1)*b (unary minus does not exist at all), and exponentiation which performs the following simplifications: x^0==>1, 1^x==>1, x^1==>x, (x^y)^z==>x^(y*z) and the expansion of sums raised to an integer power.

With these additions, the canonical form produced for multivariate polynomials can be seen to be basically a sum of products. The uniqueness of the resulting form for this

class follows, since inner terms have already been simplified, redundant summands, factors and powers have been removed, like terms combined, and the terms of sums and products put into a canonical order.

The following points about this canonical form are perhaps worth noting. First, unlike Deutsch's form, sums and products do not always contain a constant term. While this makes the simplification algorithm slightly more complex, we found it considerably easier to write the matcher using this representation. Secondly, when adding two sums together, for instance, the order of summands in the arguments is ignored; this results in an O(mn) rather than an O(m+n) algorithm, but since the terms which occur in practice are usually very short, this does not slow down the simplification algorithm appreciably. Moreover, after applying substitutions to expressions, it allows the same simplification routine to be used in ensuring that the result is still in the "simplified" form.

Finally, note that there was nothing special about addition and multiplication in the above description - they were merely operations with particular properties and identities - and the simplification process works on other functions as well, for example, conjunction and disjunction.

While the user of the algebra system can always declare new functions together with their properties, certain functions are declared for him initially. The initialized functions are +, *, - (subtraction), (exponentiation), div

(integer division) and rem (remainder). The functions div and rem are treated as any other user-declared functions would be; in their case the simplifier knows nothing about them. However, many of their properties are given as rules which are used by the theorem prover and are described in the discussion of reduction rules. For use in programs operating on lists, the functions front, back, cons, hd, tl, dest, concat and fringe are also declared initially.

An array is initially represented by a word (its name). The array's bounds are kept in its "property list" (see below). Accessing an array is done using the function acc (acc(a,i)=a[i]). Updating the array is done using the function chng; after the assignment X->A[I] to the array a its value is chng(x,i,a). This basic method of representing operations on arrays was originally proposed by McCarthy, and was used by Waldinger and Levitt. The concept of a sequence has proved important when describing programs which operate on arrays. A sequence is just a subarray of an array; the function seqof is used to construct them: seqof(a,i,j) (= a<<i,j>>) is the sequence a[i], a[i+1],...,a[j-1], a[j].

It remains to describe how the information about each function or word is stored. Conceptually, associated with each word ever seen by the system, there is a property list. For instance "*" might have the property list

[ISFUNCTION TRUE ISASSOC TRUE ISCOMM TRUE DISTRIBOVER "+"
IDENTITY 1 SYMBOLICVERSION "**" ...],

where the value of property DISTRIBOVER indicates that * distributes over +, and ** is the symbolic, term-constructing variant of *. For efficiency, this property list is implemented as a record having a finite number of components - all immediately accessible - and is kept in the word's MEANING. The first component of this record contains a bit-string representing the values of various boolean-valued properties. Thus new predicates can easily be declared and associated with words in the system.

5.1.2 Logical expressions

A logical expression is simply an expression whose values are truth-values. Two important subclasses of the logical expressions are the literals which do not contain any logical connectives other than NOT, and the relational expressions which are constructed from the operators =, /=, >, >=, <, and =<. We shall use the word "term" when referring to an expression which is not a logical expression.

All logical expressions (with two exceptions noted later) are simplified as they are constructed. For relational expressions in integer arithmetic the simplified form of equivalent expressions is unique, and is hence a canonical form. The simplification process for relational expressions is as follows:

1) If both arguments of the relation are numbers, the truth value is simply evaluated.

2) Otherwise the transformations a>b => a>=b+1, and a<b => a<=b-1 are applied (we are assuming integer arithmetic throughout).

3) The expression is then put into the form e op n, where op is one of >=, <=, = or /=, n is a number, and e is neither a number nor a sum containing a numerical summand. At this stage we would have reduced a>b, through a>=b-1 into a-b>=1, and b<a through b<=a-1 into b-a<=-1; to put these two equivalent expressions into the same form, we do the following:

4) Let revop denote the operator which satisfies e revop n if and only if n op e. Then, if e is a product whose numerical factor is less than zero, return -e revop -n. Otherwise, if e is a sum, and the first summand of -e precedes the first summand of e (in the ordering described for commutative functions), then again return -e revop -n. Otherwise leave the expression unchanged. Since a precedes b in this ordering, both the above expressions are reduced to a-b>=1.

While this procedure sounds complex, it is important for simplifying conjunctions that relational expressions should be in canonical form. The relational operators can also take subarrays as arguments: the expression a<<i,k>> >= p is equivalent to (FA j)(i=<j=<k=>a[k]>=p). In this case the only simplification done is to possibly reverse the order of the arguments. Reasoning about such expressions is then done using inference rules. For computing in domains other than integer arithmetic, some other predicates are required.

For lists, atom, null and memb are provided initially.

We come now to the logical operators. Conjunctions and disjunctions (and literals which are not relational expressions) are simplified by the same routines as arithmetic expressions. They are both associative, commutative functions having a zero and an identity. We keep expressions in conjunctive normal form, so we define v to distribute over &. The main simplifications however result from the merging done when forming the conjunction of two expressions. (The ordering used ensures that relational expressions with the same left-hand side are brought together.) Clearly P & NOT P ==> false, and P & P ==> P, but for relational expressions the transformations shown in Figure 5.1 take place as well.

```
e=m & e=n ==> false  if m/=n,
e=m & e/=m ==> false
e=m & e>=n ==> false  if m<n
e=m & e=<n ==> false  if m>n
e=m & e op n ==> e=m  otherwise

e/=m & e>=n ==> e>=n  if m<n
e/=m & e=<n ==> e=<n  if m>n
e/=m & e>=m ==> e>=m+1  (for integers)
e/=m & e=<m  ==> e=<m-1  (for integers)

e>=m & e>=n ==> e>=m  if m>=n
e>=m & e=<m ==> e=m
e>=m & e=<n ==> false  if m<n

e=<m & e=<n ==> e=<m  if m=<n
```

Figure 5.1 - Conjunction transformations

These simple rules, used with the general purpose simplifier described earlier, enable conjunctions of relational expressions to be put into their simplest form, e.g. a>=b &

a>b & a-b/=1 ==> a-b>=2. King and Deutsch, with their more specialised systems, had to write separate routines to achieve this. Two remaining rules are that a & (a v b) ==> a, and a v (a & b) ==> a.

The operators =>> (implication) and <=> (equivalence) are also available, but no simplification is done when they are applied. We will describe later how the theorem prover deals with them. The operation NOT performs the following simplifications when applied: NOT true ==> false, NOT false ==> true, NOT NOT p ==> p, NOT (a & b) ==> (NOT a) v (NOT b), NOT(a v b) ==> (NOT a) & (NOT b), NOT(a =>> b) ==> a & (NOT b), NOT(a <=> b) ==> (a v b) & (NOT a v NOT b), NOT (FA x)a ==> (EX x)NOT a, and NOT (EX x)a ==> (FA x)NOT a. It also simplifies relational expressions when applied to them, e.g. NOT(a<b) ==> a=<b ==> a-b=<0.

The quantifiers which can be used are FA, EX, ANY and ANYARR. ANY and ANYARR are variants of the epsilon operator EPS: ANYARR is used for arrays, and ANY for other kinds of objects as described previously. Again these expressions are not simplified as they are constructed, but only by the theorem prover.

5.2 Pattern matching

Clearly, any rule-driven system requires some sort of pattern matcher to control the application of rules, and ours is no exception. However, a unification algorithm has not been necessary, and we have only used a one-way pattern matcher. The simple syntactic matcher we wrote initially soon proved inadequate, and we then implemented a rather sophisticated matcher which took into account the associative, commutative and identity properties of the functions with which we were dealing. Although slower, this new matcher has proved invaluable in the subsequent development of our theorem prover. The matcher is similar to the QA4 matcher which operates on tuples and bags, but is more general in that all variables are automatically "fragment variables" and that it knows about identities.

The matcher operates on the same class of expressions as descibed above, with the exception that there are no quantifiers. It assumes that all expressions are stored in canonical form. An expression containing variables which may be instantiated is called a pattern. A substitution is a finite function mapping variables into expressions; each substitution has a unique extension which maps patterns into expressions. A one-way pattern matcher is a function

I: expressions x patterns -> substitution-sets

which satisfies

for all s in I(e,p), s(p)=e.

The equality in this condition is with respect to any laws the functions of this class of expressions may satisfy. The matcher is complete if it produces all such substitutions (ignoring the substitutions' effect on variables not contained in p).

To explain how the algorithm works, suppose we are trying to match f(s1,s2,...,sm) against f(t1,t2,...,tn) where the latter term is the pattern containing the variables to be instantiated and where e is the identity of f (if it exists). If f is not associative, we match s1 against t1, and extend the resulting substitution (if any) by recursively matching f(s2,...,sm) against f(t2,...tn). However, if f is associative, we could match any of s1, f(s1,s2), ..., f(s1,...,sm) against t1, and extend the resulting substitution by matching f(s2,...sm), f(s3,...sm), ..., f() respectively against f(t2,...tn). f() is treated as the identity e of f if one exists; f(s) is considered to be equal to s. If f is commutative as well, we could match any of s1,s2,...,sm, f(s1,s2), f(s1,s3), ..., f(s1,...sm) against t1, and the remainder of the term against f(t2,...tn). This case essentially involves finding all the subbags of a bag. In all these cases matching s1, say, against t1 may return more than one substitution, and each of these will be used in turn when matching the remainder of the terms. Notice that the operations performed in the three cases are the same, only they are applied to the

members of the successively larger sets {s1}, {e, s1, f(s1,s2), ..., f(s1,...sm)}, and {e, s1, s2, ..., sm, f(s1,s2), f(s1, s3), ..., f(s1,...,sm)}. (Actually these are sets of pairs; the first element of each pair is shown, and the second is the list of remaining arguments of the term.) We consider this method, treating the above three cases uniformly but generating a different set in each case, the most interesting aspect of the pattern matcher.

The algorithm is clearly complete with respect to the associative, commutative and identity laws. For if f(s1,s2,...sm) does match f(t1,t2,...,tn) then t1 will eventually be compared with the subterm it matches, as there is no cancellation, and the remainder of the match will be found similarly.

The actual matcher implements the above algorithm in a fairly straight-forward way. It generates the set of all the substitutions at once, rather than producing them one at a time. Since the set is usually quite small in our application, and since backtracking is quite expensive in POP-2, this method seems acceptably efficient. The POP-2 listing of the matcher is shown in Appendix 5. It uses a library program [LIB ASSOC] to represent substitutions by association sets, so that ASSOC(x,sub) = sub(x). The actual matcher also allows patterns to contain function variables which can be matched only against function constants.

Figure 5.2 shows some examples of the matcher's behaviour; a and b are constants, x, y and z are variables. Notice that 0 matches x*y in two ways since 1 is the multiplicative identity, but 0 will not match 2*x as the matcher knows nothing about the multiplicative properties of 0. More generally, the matcher does not know anything about factorization, so even 2*a*b+4*a*c will not match 2*x*y. The fact that the matcher does not know about cancellation occasionally prevents potential matches such as f(a-1,b+1)+a+b against f(x,y)+x+y from being found.

Expression	Pattern	Matches
a	x	{ (x->a) }
a	x+y	{ (x->a, y->0), (x->0, y->a) }
0	x*y	{ (x->0, y->1), (x->1, y->0) }
0	2*x	{ }
a*b	x+y	{ (x->a*b, y->0) (x->0, y->a*b) }
a+b	x*y	{ }
a+b	x+y*z	{ (x->a, y->b, z->1), (x->a, y->1, z->b), (x->b, y->a, z->1), (x->b, y->1, z->a), (x->a+b, y->0, z->1), (x->a+b, y->1, z->0) }

Figure 5.2 - Output of matcher

5.3 Automatic theorem proving

The automatic theorem prover is not intended to be a powerful, general-purpose theorem prover. Rather, it is designed to prove the simpler theorems which arise as verification conditions, leaving the more difficult ones to be proved interactively. Thus, it can prove all theorems depending only on propositional logic, and many which depend on properties of relational expressions, transitivity, reduction rules, and short sequences of inference rules. Proofs which use case-analysis, instantiations, longer sequences of inferences and so on must be done interactively.

The theorem prover can best be seen as having two phases. In the first (deterministic) phase a set of rewrite rules converts the theorem into a set of formulae of the form h1 & h2 & h3 & ... & hn => c where all the hi and c are literals (or quantified expressions). During this process any equalities in the hypotheses are eliminated by substituting one term for another throughout the formula. In the second phase, some more heuristic methods are applied to each of these formulae. In both phases the simplication routines described in Section 5.1 are used continually to simplify new conjunctions, implicit conjunctions and all other new expressions.

5.3.1 Rewriting phase

During this phase the theorem to be proved is simplified by successively applying rewrite rules until no more are applicable. To describe these rules we use a sequent calculus, writing h1,...,hn -> c to stand for the formula h1&...&hn => c. Now, letting A be a set of expressions (which represents their conjunction), and p, q and r be expressions, we have the following rules:

1a) $\dfrac{A\to p \text{ and } A\to q}{A\to p\&q}$ 1b) $\dfrac{p,q,A\to r}{p\&q,A\to r}$

2a) $\dfrac{\neg p,A,\to q}{A\to p\vee q}$ 2b) $\dfrac{p,A\to r \text{ and } q,A\to r}{p\vee q,A\to r}$

3a) $\dfrac{p,A\to q}{A\to p\Rightarrow q}$ 3b) $\dfrac{\neg p,A\to r \text{ and } q,A\to r}{p\Rightarrow q,A\to r}$

4a) $\dfrac{p,A\to q \text{ and } q,A\to p}{A \to p\Leftrightarrow q}$ 4b) $\dfrac{p,q,A\to r \text{ and } p,\ q,A\to r}{p\Leftrightarrow q,\ A \to r}$

5a) $\dfrac{A\to p(x0)}{A\to (FA\ x)p(x)}$ 5b) $\dfrac{p(x0)\to r}{(EX\ x)p(x)\to r}$

6) $\dfrac{q(x0),A \to p(x0)}{A\to p((EPS\ x)q(x))}$

7) $\dfrac{p(t1,...,tn),A \to p(s1,...,sn)}{\neg p(s1,...,sn),A \to \neg p(t1,...,tn)}$

Notes

1) These rules should be read: to prove the formula below the line it suffices to prove the formula(e) above the line.

2) The system differs from more common sequent calculi in that the consequent is an expression rather than a

disjunction of expressions.

3) In applying rule 2a) we heuristically choose the "less interesting" of p and q to negate on the grounds that there is more likely to be an inference rule applicable to q than one applicable to p. An expression is assumed to be "interesting" if its top-level function is one of the newly-declared functions EQSEQ, ISIN, ORDERED, etc. If both expressions are interesting or neither are, then p or q is chosen arbitrarily.

4) The variable x0 in rules 5a), 5b) and 6) must not occur in the formula below the line.

5) The use of the two rules

$$8a)\quad \frac{A \rightarrow p(t)}{A \rightarrow (EX\ x)p(x)} \qquad 8b)\quad \frac{p(t) \rightarrow r}{(FA\ x)p(x) \rightarrow r}$$

will be described later. Briefly, an attempt is made in stage two to instantiate the variable x of rule 8a), and the user can also apply the two rules directly.

6) The following additional rule is also applied whenever possible during this stage. It removes equalities by replacing variables equal to a term by that term.

$$9)\quad \frac{A(t) \rightarrow r(t)}{x=t, A(x) \rightarrow r(x)}$$

The system has a limited equation solver which enables this rule to be applied with expressions such as 2*a+b+3*c=4. The automatic application of this rule can be suppressed interactively if desired. A more general substitution rule

can also be applied interactively and will be described later.

7) Formulae whose consequents are inclusions involving epsilon expressions are simplified using the rules described in Section 2.4.1.

5.3.2 Heuristic phase

The original theorem has now been reduced to a set of formulae of the form h1 & ... & hn => c where each hi and c is either a literal or a quantified expression. Each of these formulae is now proved separately. To prove such a formula, each of the following methods is applied in turn.

5.3.2.1 Simplification:

This routine test whether h1&...&hn implies c as a tautology or by simple properties of the relational operators. It uses the routine described earlier for taking conjunctions of expressions. If the hypothesis h1&...&hn is false (i.e. simplifies to false) then the formula is valid; if h1&...&hn & c is false the formula is invalid; and if h1&...&hn&c equals h1&...&hn or h1&...&hn & NOT c is false, the formula is again valid. These tests include the cases that c or NOT c is one of the hi. They suffice to show, for example, that x>=1 => x>=0, and that x=<10 => x/=20.

5.3.2.2 Reduction:

If c is an equality, t1=t2, both terms t1 and t2 are transformed separately using any applicable reduction rules. If the results are identical the formula is proved; otherwise we proceed to the next step. The transformation is done by attempting to apply each reduction rule in turn at the top level. If one is applicable the procedure is repeated with the new term. If, and only if, none of the rules are applicable, each of the term's subterms is transformed using the same procedure recursively. If one of the subterms has been changed, the procedure starts again, this time applying rules at the top level only. No record is kept (at present) of which reduction rules have been applied. The "simplification" routine just described is used to check that the preconditions of the rule being applied are satisfied. As an example, using the rules of Section 4.3.1, (a*b) div b is transformed via a div 1 to a. To prevent impossible applications being attempted, reduction rules are indexed by the top-level function of their left-hand side as they are constructed, and only the rules associated with the top-level function of the current term are considered.

5.3.2.3 Instantiation:

If c is of the form (EX x1,x2,...)p(x1,x2,...) we attempt to find instantiations t1,t2,... for the variables x1,x2,... so that p(t1,t2,...) follows from the hypotheses, that is we attempt to apply rule 6a) above. p(x1,x2,...) is

typically a conjunction, and we start by forming a list of its "interesting" conjuncts (as defined above); if this is empty we use the list of all its conjunctions. We then form the set of all substitutions produced by matching each "interesting" term against each each hi, and for each substitution s we recursively attempt to prove that h1&...&hn implies s(p(x1,x2,...)) . If we have to match relational expressions they are first put into the form t op 0 since, for example, a-3>=0 matches x>=0 but a>=3 does not match x>=0. Transitivity is in the system as an inference rule which requires such an instantiation to be made for the intermediate term.

5.3.2.4 Inference rules:

This is perhaps the most general method but is very simple to describe; it is applied if all the previous methods have failed. An inference rule is applied to a formula by matching its conclusion against the conclusion, c, of the formula to be proved. For each resulting substitution, s, (provided c does not occur in s(hr)) the theorem prover is called recursively to show that h1&...&hn implies s(hr) where hr is the hypothesis of the rule. A depth counter is incremented each time this is done to prevent this search from going too deep. The heuristic that the same rule should never be used twice on the same path is also employed. This procedure is repeated for each potentially applicable inference rule - the filtering is done by applying the same indexing mechanism as before to the

conclusions of the rules - until one succeeds. Note that each substitution resulting from the match is considered in these last two methods whereas, when applying the reduction rules, only one arbitrary substitution was used.

Thus it can be seen that the theorem prover is essentially traversing an AND-OR search tree in a depth-first manner. The AND-branches arise from the natural deduction system of the first stage, and the OR-branches arise from the choice of inference rules and from the choice of substitutions when instantiating existentially quantified variables and when applying inference rules. Each node on this search tree is labelled by a list. The first element of this list is an arbitrary constant; the successors of an AND-node, [i ... j], are labelled [i ... j 1], [i ... j 2], ... respectively; the successors of an OR-node, [i ... j], are labelled [i ... j -1], [i ... j -2], ... respectively. This labelling is used when tracing the theorem prover and by the interactive commands.

5.3.3 An example

As an example of the theorem prover's behaviour, we consider the proof of the formula

$$a>b+1 \ \& \ b>c+3 \ \Rightarrow \ a>c$$

which is immediately "simplified" by the algebra system to

$$a-b>=2 \ \& \ b-c>=4 \ \Rightarrow \ a-c>=1 \ .$$

The proof is by transitivity which is in the system as the inference rule

f(x,z) <- istrans(f) & (EX y)(f(x,y) & f(y,z))

where f, x, y and z are all variables. The first phase of the theorem prover leaves the formula unchanged, as do the simplification, reduction and instantiation routines. After attempting to apply other inference rules the transitivity rule is eventually applied. Relational expressions are put into the form t op 0 when they are being matched, so f(x,z) is matched against a-c-1>=0, returning the instantiation (x->a-c-1, z->0, f-> >=). Applying this to the hypotheses of the rule yields the goal

a-b>=2 & b-c>=4 => istrans(>=) & (EX y)(a-c-1>=y & y>=0).

Now istrans is true by definition, and instantiating y yields the two matches (y->a-b-2) and (y->b-c-4). The first of these yields the goal a-c-1>=a-b-2 which simplifies to b-c>= -1 which then follows by simplification from the hypotheses.

5.4 Interactive theorem proving

If the automatic theorem prover fails to find a proof it can ask the user for help. The user can then investigate why the theorem prover failed and try to lead it to a proof (as a proof-checker), or else abort the proof, all using the commands described below. The user can also interrupt the theorem prover while it is running if he desires, see what it is doing, and either continue in interactive mode from there or let it resume running. To use the interactive facilities effectively it is necessary to have a general

idea of how the automatic theorem prover works and to have available a list of all the relevant rules known to the system.

5.4.1 Interactive commands

It is useful to divide the commands available into three categories: informative, control and advice. The informative commands are used to display the current theorem being worked on, the top-level theorem, or the hypotheses or conclusions of either of these. Since the whole power of the POP-2 language is available while typing interactive commands, any other aspects of the program's state can also be examined; this ability has proved very useful while developing the program. The control commands are used to move around the AND-OR search space of the problem, and to regulate the degree to which the theorem prover either runs freely or under the user's control. The advice commands are the most important in actually aiding the theorem prover to find a proof; they are used to fill gaps in its knowledge by adding hypotheses, axioms or lemmas, to narrow its search space by eliminating alternatives, and to choose values for variables to be instantiated. We now list the commands available to the user. Each command or sequence of commands, except for control commands and those marked with an asterisk, is terminated by the user typing GOON.

5.4.1.1 Informative

PRTHM: Prints the theorem currently being proved, possibly a subgoal of the top-level theorem.

PPR(item); Prints the item in a readable format. The item is normally one of HYPS, GOAL, TOPHYP, TOPGOAL or CUNODE which refer respectively to the current hypotheses and goal, the top-level hypothesis and goal, and the label of the current node in the search tree.

TRACE n; A trace of the proof process, of successively increasing detail, is printed on the current output device by setting n to 0, 1 or 2.

5.4.1.2 Control

ASSUME: The current theorem is assumed valid, and is saved on a global list, THMSPOST, to be proved later.

ASSUMENS; The same as ASSUME but the current theorem is not saved for later.

ASSUMEAND; The current theorem is assumed valid, as are all its brothers at the AND-node immediately above the current node, and all these theorems are saved on THMSPOST to be proved later.

FAIL; The current theorem is assumed invalid, and the theorem prover returns to the previous node.

FAILTO node; The theorem at node is assumed to be invalid, and the theorem prover returns to the previous node. The node must be higher up the search tree on the current path; it is written as a list, e.g. [10 2 -4 1 -3].

FAILOR; The current theorem is assumed to be invalid as are all its brothers at the OR-node immediately above the current node, and the theorem prover returns to that node.

STEP; The theorem prover is forced into a completely interactive mode; after each step it prints its current position and goal and waits for a new command from the user.

GO; Cancels the STEP command, allowing the theorem prover to run automatically again.

RESTART; A safety device: it allows the user to start again with the top-level theorem.

5.4.1.3 Advice

LEMMA(hypotheses,conclusion,name);

Try to prove the conclusion from the hypotheses. If successful, add it as a new inference rule with the given name. For example,

LEMMA([% X REM 2 /= 0 %], (X-1)REM 2 = 0, "REM1");

RLEMMA(hypotheses,leftside,rightside);

Try to prove that the hypotheses imply the other two expressions are equal. If successful, add the corresponding new reduction rule.

AXIOM rule; Add the new rule to the present ones without trying to prove it, but save it on THMSPOST to be proved later.

DEDUCE expression;

Try to prove the logical expression from the current hypotheses (HYPS) and then add it to them.

ADD expression; The logical expression is temporarily added to the current hypotheses and the theorem that the current hypotheses imply the expression is saved.

ADDNS expression; Like ADD but the theorem that the current hypotheses imply the expression is not saved.

DEL expression; The expression is temporarily removed from the current hypotheses.

USE rulename; The inference with the given name is applied at the next opportunity, for example, USE REM1;

CASES expression-list; (*)

Each of the cases is successively assumed and temporarily added to the current hypotheses. The cases must be exhaustive with respect to the hypotheses. For example, CASES [% X=<N-1, X=N %];

INTERM term; (*) Tries to prove both f(a,term) and f(term,b) where the current goal is f(a,b) and f is a transitive operator.

LETEX(var,term); Assigns the term to the existentially quantified variable var of the goal, thereby implementing rule 8a).

LETFA(var,term); Assuming one of the hypotheses contains a universally quantified variable var, instantiate var to term in it, thereby implementing rule 8b). If there is more than one such hypothesis, typing LETFA(var,n,term) instantiates the nth

hypothesis.

LET var1 BE term1, ... varn BE termn IN rulename;

Use the named inference rule with the variables instantiated as shown (cf. POP-2 partial application). This is sometimes necessary to overcome weaknesses in the matcher.

REWRITE; (*) Applies the rewriting rules 1) to 9) to remove any logical connectives which have appeared in the hypotheses and to substitute for any equalities which have appeared.

CONTRAPOS n; (*) Prove the contrapositive form of the current theorem, negating the nth hypothesis. For example, if the current theorem is a[i]>a[i+1] & i<j => NOT ORDERED(a,i,j), after executing CONTRAPOS 1; the theorem would become i<j & ORDERED(a,i,j) => a[i]≤a[i+1], which is easier to prove from the definition of ORDERED given in Appendix 2. CONTRAPOS 0 invokes proof by contradiction by converting the theorem A=>p to NOT p,A=>false.

DOSUBST(expr1,expr2,expr3); Substitutes expr1 for all occurrences of expr2 in expr3; expr3 would normally be GOAL or HYPS, and expr1 must be provably equal to expr2. For example, DOSUBST(A+B, C2, GOAL) -> GOAL;

SIMPGOAL; This transforms the goal by applying reduction rules. At one stage this was done automatically and sometimes prevented proofs being found, so it is now done only when requested.

SIMPHYPS; Similiarly, this applies the reduction rules to all the expressions in the hypotheses.

SIMPHYP expression; Applies the reduction rules to the particular expression in the hypotheses.

INDUCT var; Prove the current theorem by mathematical induction on the variable var, e.g. INDUCT M;

INDUC2 var; Prove the current theorem by course-of-values induction on the variable var. These two rules are required for proving derived rules rather than programs where the induction has been done by the proof method.

INDLEMMA(hypotheses,conclusion,name,var);

Try to prove that the hypotheses imply the conclusion by mathematical induction on var. Then add it as a new inference rule with the given name.

SWEAT n; Temporarily increases the depth to which the proof can go by n.

5.4.2 Another example

As a typical example of these rules, we reproduce one proof done while verifying the sorting program of Section 6.3. Commands typed by the user appear in lower case, and explanatory comments are indented. The hypothesis of a theorem is represented by the list of its conjuncts. Proofs of particular goals and subgoals are shown as tree structures, with the rule name or routine at the root followed by the instantiation used, and a description of how the remaining subgoals were proved; SIMP, CONJ and INST

refer to simplification, conjunction and instantiation respectively.

```
FAILED [ 5 1 5] ORDERED(XCHNG(A,I,K)[ 1,I])
>: prthm;
[(I >=  1) (A[I,-1+J]) >= A[K]) (A[I,-1+J]) >= A[-1+I])
(I-J =< -3) (I-K =<  0) (J-K >= 1 )
PERM(A[ 1,-1+J],A0[ 1,-1+J]) ORDERED(A[ 1,-1+I])]
    ==>
ORDERED([XCHNG(A,I,K)[ 1,I]):
:
: cases [% "i"=1, "i">1 %];
```

This sort of case analysis is usually required when using our recursive definitions to prove an array segment is ordered.

```
CASE PROVED [ORD1 [ (XJ. 1) (XI. 1) (XA.XCHNG(A,1,K))] SIMP]
FAILED [ 5 1 5 2] ORDERED(XCHNG(A,I,K)[ 1,I])
```

The system has proved the first case (I=1) using the rule ORD1 (see Appendix 2) with the instantiation shown. The second case now remains to be proved.

```
>: use ord12;
: goon
```

ORD12 is a counting-down property of ORDERED. It is applied here since we are given ORDERED(A[1,I-1]) as a hypothesis.

```
FAILED [ 5 1 5 2 -1 2]
       (XCHNG(A,I,K)[I]-XCHNG(A,I,K)[-1+I] >=  0)

>: simpgoal;

(A[K]-A[-1+I] >=  0):
:
: use xf6;
: goon
```

XF6 is a rule which proves that an array element has a property if it is in an array segment all of whose members have that property.

```
FAILED [ 5 1 5 2 -1 2 -1] EX([XI XK],((A[XI,XK] >= A[-1+I])
&  (XI-K =<  0)  &  (XK=K >=  0)), 3)

>: sweat 1;
: goon
```

Because the default depth is normally 1, and using the instantiation routine increments the depth counter, the instantiation is not automatically attempted. However, after temporarily increasing the depth bound, the correct instantiation is found.

```
FAILED [ 5 1 5 2 -1 3] ORDERED(XCHNG(A,I,K)[ 1,-1+I])
```

This is the third hypothesis of the rule ORD12.

```
>: simpgoal;

ORDERED(A[ 1,-1+I]):
:
: goon

CASE PROVED [ORD12 [ (XJ.I) (XI. 1) (XA.XCHNG(A,I,K))] [CONJ
SIMP [XF6 [ (XK.-1+J) (XI.I)] [INST [ (XK.-1+J) (XI.I) ]
[CONJ SIMP SIMP SIMP]]] SIMP]]
CASES PROVED [(I = 1) (I >= 2)]
: goon

[ 5 1 5]  PROVED
```

5.5 Discussion

The top-down tree searching method of proof described above is basically the same as that used by Waldinger and Levitt, Bledsoe and Bruell, and Milner. It contrasts with the approach taken by King, Deutsch, and resolution systems of negating the conclusion and trying to derive a contradiction. The direct approach seems preferable in that it is more natural, easier for the user to understand, and hence easier for him to give appropriate advice. A possible disadvantage may arise when the user knows how the proof should go, wants to do it in a bottom-up fashion, but finds he is being driven by the theorem prover rather than vice-versa. In our system such bottom-up proofs can be done by making deductions from the hypotheses, and by proving

lemmas to be treated as reduction rules and then calling SIMPHYPS

In several systems, including ours, Waldinger and Levitt's, Suzuki's and the new LCF (Milner 1975), it is possible to introduce derived inference rules. Good, London and Bledsoe can define reduction rules to describe their newly-introduced functions. Weir and Burstall (1972) also used derived rules (macro-inferences) in a resolution-based proof checker for program correctness. A restriction of our system is the limited language in which rules can be written; a more powerful language (such as QA4) would be useful. But whereas our rules are simple logical formulae which can be proved, the QA4 rules are themselves programs and hence harder to verify. Only in our system and Milner's are derived rules proved from definitions, though Moore's (1974) LISP theorem prover uses theorems which have been previously proved as lemmas.

Much of the "knowledge" our system has is embodied in its rules. While this is a flexible system it can be rather inefficient. The main problem is that a large number of obviously useless rules are optimistically applied by the theorem prover. An indexing scheme which considered the expressions in the hypotheses as well as the conclusion would greatly reduce this branching ratio and improve the speed, if not the power, of the theorem prover. There is a clear trade-off here between (i) storing all rules concerning the functions involved, which enables some

programs to be proved automatically but slowly, (ii) storing only some of them (the relevant ones), and (iii) not storing any rules, which means more interaction (to provide the rules) but faster execution. Roughly speaking, we take the first alternative, Suzuki the second, and Good, London and Bledsoe the third. Our reasoning in taking this alternative is that the system itself should be able to choose the right property (i.e. rule) to use, since otherwise the user would either have to delete rules given previously or to give the same rule more than once in the course of proving a single program.

Chapter 6. EXAMPLES OF PROGRAM BEHAVIOUR

In this chapter we shall study the verifier's performance on several typical programs. The progams considered are the 91-function, a program to test whether one array is a subarray of another, and an insertion sort program taken from King's thesis. In each case, we start by showing the specified program which is input to the verifier, followed by the output of the verifier. The user's commands are shown in lower case to distinguish them from the verifier's output which is in upper case. Explanatory comments have been added to the proof, indented and in lower case. In Section 6.4 we show two different proofs of an inference rule involving the predicate sorted.

In these examples the variable MAXDEPTH refers to the maximum depth (in terms of the number of inference rules applied) to which the proof can go, though this can be overridden at the user's command. The value of CONDPROVE determines whether proofs are attempted at conditionals or not, and TRACE indicates the detail to which the proof is given. The times shown are in seconds. The reader should also remember that in POP-2 the truthvalues TRUE and FALSE have the values 1 and 0 respectively.

6.1 The 91-function

We have already seen this program in Section 2.5.1. The verifier is able to prove it is correct without any assistance from the user. No inference rules are used, but only the built-in routines for reasoning about relational expressions and substitution of equalities.

```
FUNCTION FN91 N => R;
  IF N>100 THEN N-10 ELSE FN91(FN91(N+11)) CLOSE -> R;
END;


VARS N0;
[N0]->INITARGS;

ISTAT BODY REC FN91;
VIRT  IF N>100 THEN N-10 ELSE 91 CLOSE -> R;
UNDER TRUE;
WRT   [R]
```

Proof

```
FN91   14-4-1975
MAXDEPTH= 1  CONOPROVE= 1  TRACE= 1

COMPILE TIME = 0.753

STARTING NEW PROOF FROM FN91BEG
```

The function FN91 has labels FN91BEG and FN91END inserted at the beginning and the end of its body.

```
. . . . . .

RUN ACTUAL PROGRAM

AT COND  (N >=  101)
```

The actual program has reached the conditional in FN91. The verifier now tries to prove either that the condition or its negation follows from the path condition. As the initial path condition is TRUE, its list of conjuncts is NIL.

```
ASSUME TRUE
```

This refers to the preceding condition. The path terminates immediately, and the alternative path is then taken.

```
ASSUME FALSE
```

The inner call of function FN91 has just been entered, so the precondition of its inclusion statement is being tested. (N=<100) is the current path condition.

```
TEST PRECONDS OF INCLUSION AT FN91BEG
 1
FROM PATH CONOITION
(N =<  100)
OK   SIMP
```

SIMP indicates that the condition was proved by simplification. The virtual program is now executed. The test N>=90 corresponds to N+11>=101.

```
AT COND  (N >=  90)
ASSUME TRUE
```

The outer call of FN91 is now entered with path condition (N>=90 & N=<100), and with state vector N->N+1 (=N+11-10).

```
TEST PRECONDS OF INCLUSION AT FN91BEG
 1
FROM PATH CONDITION
((N >=  90)  &  (N =<  100))
OK   SIMP

AT CONO  (N >=  100)
ASSUME TRUE
ASSUME FALSE
ASSUME FALSE
```

That is, assume N=<89 (N+11=<100). The outer call of FN91 is now entered with path condition N=<89, and with state vector N->91.

```
TEST PRECONOS OF INCLUSION AT FN91BEG
 1
FROM PATH CONDITION
(N =<  89)
OK   SIMP

AT COND   0
MUST BE FALSE,  FROM PATH CONDITION
(N =<  89)

RUN VIRTUAL PROGRAM
```

All the paths of the actual program have terminated, and the virtual program is now executed.

```
AT COND  (N >=  101)
ASSUME TRUE
ASSUME FALSE
```

All the paths of the virtual program have now terminated also. Each pair of states is now considered in turn.

```
FROM PATH CONDITION (N >=  101)
     AND (N >=  101)
    I.E. (N >=  101)
```

The first two expressions are the path conditions of the actual and virtual programs respectively; the third expression is their conjunction.

```
PROVE INCLUSION
R:   -10+N       -10+N   OK  SIMP
```

The equality N-10=N-10 has simplified to TRUE.

```
STACKS:  NIL    NIL   OK
END POINTS:  FN91END  FN91END
```

The program is now considering the next pair of states. It tries to establish the inconsistency of the two path conditions by proving that the negation of a conjunct of the virtual program's path condition follows from the actual program's path condition, i.e., that N>=101 implies NOT N=<100.

```
THE ACTUAL-VIRTUAL PAIR:
(N >=  101)
(N =<  100)
IS INCONSISTENT:  SIMP

THE ACTUAL-VIRTUAL PAIR:
(N =  100)
(N >=  101)
IS INCONSISTENT:  SIMP

FROM PATH CONDITION (N =  100)
     AND (N =<  100)
    I.E. (N =  100)
PROVE INCLUSION
R:   -9+N        91   OK  SIMP
```

The theorem was trivially proved by substituting the equality of the hypotheses into the goal. The remainder of the theorems are all proved directly by simplification.

```
STACKS:  NIL    NIL   OK
END POINTS:  FN91END  FN91END
```

```
THE ACTUAL-VIRTUAL PAIR:
((N >=  90)  &  (N =<  99))
(N >=  101)
IS INCONSISTENT:  SIMP

FROM PATH CONDITION ((N >=  90)  &  (N =<  99))
     AND (N =<  100)
    I.E. ((N >=  90)  &  (N =<  99))
PROVE INCLUSION
R:    91       91   OK  SIMP
STACKS:  NIL    NIL   OK
END POINTS:  FN91END  FN91END

THE ACTUAL-VIRTUAL PAIR:
(N =<  89)
(N >=  101)
IS INCONSISTENT:  SIMP

FROM PATH CONDITION (N =<  89)
     AND (N =<  100)
    I.E. (N =<  89)
PROVE INCLUSION
R:    91       91   OK  SIMP
STACKS:  NIL    NIL   OK
END POINTS:  FN91END  FN91END

RUN TIME = 5.536

THEOREMS PENDING: NIL
```

6.2 A matching program

This program determines whether the array B occurs as a subarray of the array A. The verifier can prove this program completely automatically with the exception of one application of CONTRAPOS invoked by the user.

```
FUNCTION MATCHV A M B N => BISINA;
VARS I J ;
 FALSE->BISINA;
 0->I;
$LOOPI:
 LOOPIF I=< M-N THEN
    1->J;
  $LOOPJ:
    LOOPIF J=<N THEN
       IF A\(I+J)/=B\(J) THEN GOTO BREAKJ CLOSE;
       J+1->J;
    CLOSE;
  $ENDLOOPJ:
    TRUE->BISINA; GOTO BREAKI;
  $BREAKJ:
    I+1->I;
 CLOSE;
$BREAKI:
END;


DECARRAY A [1 M];
DECARRAY B [1 N];
[%A,"M",B,"N"%]->INITARGS;

ISTAT BODY MATCHV;
VIRT  IF ISIN(B<<1,N>>,A<<1,M>>)
      THEN TRUE ELSE FALSE CLOSE -> BISINA;
UNDER 0=<N &  N=<M;
WRT   [BISINA];

ISTAT LOOPI TO [BREAKI]
VIRT  IF ISIN(B<<1,N>>,A<<I+1,M>>)
      THEN TRUE->BISINA;
      ELSE M-N+1->I CLOSE;
      UNDER 0=<I &  I=<M-N+1 &  0=<N;
WRT   [BISINA];

ISTAT LOOPJ TO [ENDLOOPJ BREAKJ];
VIRT  IF EQSEQ(B<<J,N>>,A<<I+J,I+N>>)
      THEN N+1->J; GOTO ENDLOOPJ
      ELSE GOTO BREAKJ CLOSE;
UNDER  1=<J &  J=<N+1;
WRT   [I];
```

Proof

```
MATCHV   13-5-1975
MAXDEPTH= 1  CONDPROVE= 0  TRACE= 0

COMPILE TIME = 8.169

STARTING NEW PROOF FROM MATCHVBE

        Each program FOO implicitly contains labels called
        FOOBEG  and FOOENO at the beginning and end of its
        body.  In this case the POP-2 system truncates the
        name MATCHVBEG to MATCHVBE.
    . . . . . .

RUN ACTUAL PROGRAM

TEST PRECONDS OF INCLUSION AT LOOPI
((N >=  0)  &  (M-N >= -1))
FROM PATH CONDITION
((N >=  0)  &  (M-N >=  0))
OK  [CONJ SIMP SIMP]

AT CONO  ISIN(B[ 1,N],A[ 1,M])
ASSUME TRUE
ASSUME FALSE

RUN VIRTUAL PROGRAM

AT CONO  ISIN(B[ 1,N],A[ 1,M])
ASSUME TRUE
ASSUME FALSE

ASSUMING ((N >=  0)  &  (M-N >=  0)  &  ISIN(B[ 1,N],A[ 1,M]))
     AND ((N >=  0)  &  (M-N >=  0)  &  ISIN(B[ 1,N],A[ 1,M]))
    I.E. ((N >=  0)  &  (M-N >=  0)  &  ISIN(B[ 1,N],A[ 1,M]))
PROVE INCLUSION
BISINA:    1       1   OK  SIMP
STACKS:  NIL    NIL   OK
END POINTS:  MATCHVEN  MATCHVEN

THE ACTUAL-VIRTUAL PAIR:
((N >=  0)  &  (M-N >=  0)  &  ISIN(B[ 1,N],A[ 1,M]))
((N >=  0)  &  (M-N >=  0)  &  NOT(ISIN(B[ 1,N],A[ 1,M])))
IS INCONSISTENT:  SIMP

THE ACTUAL-VIRTUAL PAIR:
((N >=  0)  &  (M-N >=  0)  &  NOT(ISIN(B[ 1,N],A[ 1,M])))
((N >=  0)  &  (M-N >=  0)  &  ISIN(B[ 1,N],A[ 1,M]))
IS INCONSISTENT:  SIMP

ASSUMING ((N >=  0) & (M-N >=  0) & NOT(ISIN(B[ 1,N],A[ 1,M])))
     AND ((N >=  0) & (M-N >=  0) & NOT(ISIN(B[ 1,N],A[ 1,M])))
    I.E. ((N >=  0) & (M-N >=  0) & NOT(ISIN(B[ 1,N],A[ 1,M])))
PROVE INCLUSION
BISINA:    0       0   OK  SIMP
STACKS:  NIL    NIL   OK
```

```
ENO POINTS:  MATCHVEN  MATCHVEN

STARTING NEW PROOF FROM LOOPI
```

The first inclusion statement has been verified and the system is now starting to verify the one whose start point is LOOPI.

. . . . . .

```
RUN ACTUAL PROGRAM

AT CONO  (M-N-I >=  0)
ASSUME TRUE

TEST PRECONOS OF INCLUSION AT LOOPJ
(N >=  0)
FROM PATH CONOITION
((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0))
OK  SIMP

AT CONO  EQSEQ(A[ 1+I,N+I],B[ 1,N])
ASSUME TRUE
ASSUME FALSE

TEST PRECONOS OF INCLUSION AT LOOPI
((N >=  0)  &  (I >= -1)  &  (M-N-I >=  0))
FROM PATH CONOITION
((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ(A[
1+I,N+I],B[ 1,N])))
OK  [CONJ SIMP SIMP SIMP]

AT CONO  ISIN(B[ 1,N],A[ 2+I,M])
ASSUME TRUE
ASSUME FALSE
ASSUME FALSE

RUN VIRTUAL PROGRAM

AT CONO  ISIN(B[ 1,N],A[ 1+I,M])
ASSUME TRUE
ASSUME FALSE

ASSUMING ((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  EQSEQ(A[
1+I,N+I],B[ 1,N]))
    AND ((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  ISIN(B[
1,N],A[ 1+I,M]))
   I.E. ((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  EQSEQ(A[
1+I,N+I],B[ 1,N])  &  ISIN(B[ 1,N],A[ 1+I,M]))
PROVE INCLUSION
BISINA:    1       1   OK  SIMP
STACKS:  NIL    NIL   OK
ENO POINTS:  BREAKI  BREAKI

THE ACTUAL-VIRTUAL PAIR:
((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  EQSEQ(A[ 1+I,N+I],
B[ 1,N]))
((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  NOT(ISIN(B[
```

```
1,N],A[ 1+I,M])))
IS INCONSISTENT:  [ISIN1 [ (XM.M) (XK. 1+I) (XB.A) (XJ.N)
(XI. 1) (XA.B)] [CONJ SIMP SIMP]]

ASSUMING ((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ
(A[ 1+I,N+I],B[ 1,N]))  &  ISIN(B[ 1,N],A[ 2+I,M]))
     AND ((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  ISIN(B[
1,N],A[ 1+I,M]))
    I.E. ((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ
(A[ 1+I,N+I],B[ 1,N]))  &  ISIN(B[ 1,N],A[ 1+I,M])  &  ISIN(B[
1,N],A[ 2+I,M]))
PROVE INCLUSION
BISINA:    1        1   OK  SIMP
STACKS:  NIL    NIL   OK
END POINTS:  BREAKI  BREAKI

THE ACTUAL-VIRTUAL PAIR:
((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ(A[
1+I,N+I],B[ 1,N]))  &  ISIN(B[ 1,N],A[ 2+I,M]))
((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  NOT(ISIN(B[
1,N],A[ 1+I,M])))
IS INCONSISTENT:  [ISIN2 [ (XM.M) (XK. 1+I) (XB.A) (XJ.N)
(XI. 1) (XA.B)] SIMP]

THE ACTUAL-VIRTUAL PAIR:
((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ(A[
1+I,N+I],B[ 1,N]))  &  NOT(ISIN(B[ 1,N],A[ 2+I,M])))
((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  ISIN(B[ 1,N],A[
1+I,M]))
IS INCONSISTENT:  [ISIN4 [ (XM.M) (XK. 2+I) (XB.A) (XJ.N)
(XI. 1) (XA.B)] [CONJ SIMP SIMP SIMP]]

ASSUMING ((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ
(A[ 1+I,N+I],B[ 1,N]))  &  NOT(ISIN(B[ 1,N],A[ 2+I,M])))
     AND ((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  NOT(ISIN(
B[ 1,N],A[ 1+I,M])))
    I.E. ((N >=  0)  &  (I >=  0)  &  (M-N-I >=  0)  &  NOT(EQSEQ
(A[ 1+I,N+I],B[ 1,N]))  &  NOT(ISIN(B[ 1,N],A[ 1+I,M]))  &
 NOT(ISIN(B[ 1,N],A[ 2+I,M])))
PROVE INCLUSION
BISINA:   BISINA      BISINA   OK  SIMP
STACKS:  NIL    NIL   OK
END POINTS:  BREAKI  BREAKI

THE ACTUAL-VIRTUAL PAIR:
((N >=  0)  &  (I >=  0)  &  (M-N-I = -1))
((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  ISIN(B[ 1,N],A[
1+I,M]))
IS INCONSISTENT:  [ISIN20 [ (XL.-1+N+I) (XK. 1+I) (XB.A) (XJ.N)
(XI. 1) (XA.B)] SIMP]

ASSUMING ((N >=  0)  &  (I >=  0)  &  (M-N-I = -1))
     AND ((N >=  0)  &  (I >=  0)  &  (M-N-I >= -1)  &  NOT(ISIN(
B[ 1,N],A[ 1+I,M])))
    I.E. ((N >=  0)  &  (I >=  0)  &  (M-N-I = -1)  &  NOT(ISIN(B
[ 1,N],A[ 1+I,M])))
PROVE INCLUSION
```

```
BISINA:   BISINA      BISINA   OK  SIMP
STACKS:  NIL    NIL   OK
ENO POINTS:  BREAKI  BREAKI

STARTING NEW PROOF FROM LOOPJ

AT CONO  (M-N >=  0)
ASSUME TRUE
    . . . . . .

RUN ACTUAL PROGRAM
AT CONO  (N-J >=  0)
ASSUME TRUE

AT CONO  (A[I+J]-B[J] /=  0)
ASSUME TRUE
ASSUME FALSE

TEST PRECONOS OF INCLUSION AT LOOPJ
((J >=  0)  &  (N-J >=  0))
FROM PATH CONOITION
((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0))
OK  [CONJ SIMP SIMP]

AT CONO  EQSEQ(A[ 1+I+J,N+I],B[ 1+J,N])
ASSUME TRUE
ASSUME FALSE
ASSUME FALSE

RUN VIRTUAL PROGRAM

AT CONO  EQSEQ(A[I+J,N+I],B[J,N])
ASSUME TRUE
ASSUME FALSE

ASSUMING ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] /=  0))
     AND ((J >=  1)  &  (N-J >= -1)  &  EQSEQ(A[I+J,N+I],B[J,N]))
    I.E. ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] /=  0)
 &  EQSEQ(A[I+J,N+I],B[J,N]))
PROVE INCLUSION
I:   I      I   OK  SIMP
STACKS:  NIL    NIL   OK
ENO POINTS:  BREAKJ  ENDLOOPJ
```

The system realizes that something is wrong as the two end points are different so it goes back and again tries to show that the two sets of conditions are inconsistent (without initial success).

```
FAILED [ 37] NOT(EQSEQ(A[I+J,N+I],B[J,N]))

>prthm;

[(J >=  1) (N-J >=  0) (A[I+J]-B[J] /=  0)]
    ==>
```

```
NOT(EQSEQ(A[I+J,N+I],B[J,N]))

contrapos 3;

FAILED [ 37 1] (A[I+J]-B[J] =  0)

>ppr(hyps);
[(J >=  1) (N-J >=  0) EQSEQ(A[I+J,N+I],B[J,N])]

sweat 1;
goon
CONTRAPO PROVED [EQSEQ3 [ (XJ.N) (XM.N+I)] [INST [ (XJ.N)
(XM.N+I)] [CONJ SIMP SIMP]]]
goon

[ 37]  PROVED

THE ACTUAL-VIRTUAL PAIR:
((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] /=  0))
((J >=  1)  &  (N-J >= -1)  &  EQSEQ(A[I+J,N+I],B[J,N]))
IS INCONSISTENT:  [CONTRAPO  3]

ASSUMING ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] /=  0))
     AND ((J >=  1)  &  (N-J >= -1)  &  NOT(EQSEQ(A[I+J,N+I],B[J,
N])))
    I.E. ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] /=  0)
 &  NOT(EQSEQ(A[I+J,N+I],B[J,N])))
PROVE INCLUSION
I:   I       I   OK  SIMP
STACKS:  NIL     NIL   OK
END POINTS:  BREAKJ  BREAKJ

ASSUMING ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0)
 &  EQSEQ(A[ 1+I+J,N+I],B[ 1+J,N]))
     AND ((J >=  1)  &  (N-J >= -1)  &  EQSEQ(A[I+J,N+I],B[J,N]))
    I.E. ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0)
 &  EQSEQ(A[ 1+I+J,N+I],B[ 1+J,N])  &  EQSEQ(A[I+J,N+I],B[J,N]))
PROVE INCLUSION
I:   I       I   OK  SIMP
STACKS:  NIL     NIL   OK
END POINTS:  ENDLOOPJ  ENDLOOPJ

THE ACTUAL-VIRTUAL PAIR:
((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0)  &  EQSEQ(A[
1+I+J,N+I],B[ 1+J,N]))
((J >=  1)  &  (N-J >= -1)  &  NOT(EQSEQ(A[I+J,N+I],B[J,N])))
IS INCONSISTENT:  [EQSEQ2 [ (XM.N) (XK.J) (XB.B) (XJ.N+I)
(XI.I+J) (XA.A)] [CONJ SIMP SIMP SIMP SIMP]]

THE ACTUAL-VIRTUAL PAIR:
((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0)  &  NOT(EQSEQ(A
[ 1+I+J,N+I],B[ 1+J,N])))
((J >=  1)  &  (N-J >= -1)  &  EQSEQ(A[I+J,N+I],B[J,N]))
IS INCONSISTENT:  [EQSEQ4 [ (XM.N) (XK. 1+J) (XB.B) (XJ.N+I)
(XI. 1+I+J) (XA.A)] [CONJ SIMP SIMP SIMP]]
```

```
ASSUMING ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0)
 &  NOT(EQSEQ(A[ 1+I+J,N+I],B[ 1+J,N])))
     AND ((J >=  1)  &  (N-J >= -1)  &  NOT(EQSEQ(A[I+J,N+I],B[J,
N])))
    I.E. ((J >=  1)  &  (N-J >=  0)  &  (A[I+J]-B[J] =  0)
 &  NOT(EQSEQ(A[ 1+I+J,N+I],B[ 1+J,N]))  &  NOT(EQSEQ(A[I+J,N+I],
B[J,N])))
PROVE INCLUSION
I:   I       I   OK  SIMP
STACKS:  NIL     NIL   OK
END POINTS:  BREAKJ  BREAKJ

ASSUMING ((J >=  1)  &  (N-J = -1))
     AND ((J >=  1)  &  (N-J >= -1)  &  EQSEQ(A[I+J,N+I],B[J,N]))

    I.E. ((J >=  1)  &  (N-J = -1)  &  EQSEQ(A[I+J,N+I],B[J,N]))
PROVE INCLUSION
I:   I       I   OK  SIMP
STACKS:  NIL     NIL   OK
END POINTS:  ENDLOOPJ  ENDLOOPJ

THE ACTUAL-VIRTUAL PAIR:
((J >=  1)  &  (N-J = -1))
((J >=  1)  &  (N-J >= -1)  &  NOT(EQSEQ(A[I+J,N+I],B[J,N])))
IS INCONSISTENT:  [EQSEQ1 [ (XM.-1+J) (XK.J) (XB.B) (XJ.-1+I+J)
(XI.I+J) (XA.A)] [CONJ SIMP SIMP]]

RUN TIME = 67.661

THEOREMS PENDING: NIL
```

6.3 A sorting program

This program is from King's thesis, although his system was unable to prove it. Our proof includes the fact that the final array is a permutation of the original one. The proof requires advice from the user in several places but in many cases it suffices to tell the automatic theorem prover to work harder. Most of the substantative advice given by the user consists of the commands CASES and USE.

```
FUNCTION K9 A N;
VARS I J K X;
   ASSERT A=A0 & 2=<N;
   1->I;
$L1:
   LOOPIF I<N THEN
      A\I->X; I->K; I+1->J;
   $L2:
      ASSERT 1=<I & I=<K & K<J & J=<N+1
           & I<N
           & X=A\K
           & (I=1 OR A\(I-1) =< A<<I,N>> )
           & A<<I,J-1>> >= X
           & ORDERED(A<<1,I-1>>)
           & PERM(A<<1,N>>,A0<<1,N>>);

      LOOPIF J=<N THEN
         IF X>A\J THEN A\J->X; J->K CLOSE;
         J+1->J;
      CLOSE;
      A\I->A\K; X->A\I; I+1->I;
   CLOSE;
$OUT:
   ASSERT ORDERED(A<<1,N>>) & PERM(A<<1,N>>,A0<<1,N>>);
END;

VARS N0;
DECARRAY A0 [1 N0];
[% A0,"N0" %]->INITARGS;
```

Proof

```
K9   13-5-1975
MAXOEPTH= 1  CONOPROVE= 0  TRACE= 0

COMPILE TIME = 5.442

STARTING NEW PROOF FROM K9BEG
    . . . . . .

AT CONO  (N >=  2)
ASSUME TRUE
TEST ASSERTION AT L2
((N >=  2)  &  (A[ 1, 1] >= A[ 1])  &  PERM(A[ 1,N],A0[ 1,N])
 &  OROEREO(A[ 1, 0]))
FROM PATH CONOITION
((N >=  2)  &  (A0 = A))
FAILEO [ 1 2] (A[ 1, 1] >= A[ 1])

>sweat 1;
goon

[ 1 2]  PROVEO
OK  [CONJ SIMP [XF1 [ (XZ.A[ 1]) (XJ. 1) (XI. 1) (XA.A) (XF.>=)]
[CONJ [XF0 [ (XZ.A[ 1]) (XJ. 0) (XI. 1) (XA.A) (XF.>=)] SIMP]
SIMP]] [PERM1 [ (XM.N) (XK. 1) (XB.A) (XJ.N) (XI. 1) (XA.A)]
REOUCTN] [ORO1 [ (XJ. 0) (XI. 1) (XA.A)] SIMP]]
ASSUME FALSE

TEST ASSERTION AT OUT
(PERM(A[ 1,N],A0[ 1,N])  &  OROEREO(A[ 1,N]))
FROM PATH CONOITION
 0
OK  [CONJ SIMP SIMP]

STARTING NEW PROOF FROM L2

AT CONO  (N0 >=  2)
ASSUME TRUE
    . . . . . .

AT CONO  (N-J >=  0)
ASSUME TRUE

AT CONO  (X-A[J] >=  1)
ASSUME TRUE

TEST ASSERTION AT L2
((I >=  1)  &  (A[I,J] >= A[J])  &  (N-I >=  1)  &  (N-J >=
 0)  &  (I-J =<  0)  &  PERM(A[ 1,N],A0[ 1,N])  &  OROEREO(A[
1,-1+I])  &  ((I =  1)  OR  (A[I,N] >= A[-1+I])))
FROM PATH CONOITION
((I >=  1)  &  (A[I,-1+J] >= X)  &  (N-I >=  1)  &  (N-J >=
 0)  &  (I-K =<  0)  &  (J-K >=  1)  &  (X-A[J] >=  1)  &
 (X-A[K] =  0)  &  PERM(A[ 1,N],A0[ 1,N])  &  OROEREO(A[ 1,-1+I])
  &  ((I =  1)  OR  (A[I,N] >= A[-1+I])))
```

```
FAILED [ 3 1 2] (A[I,J] >= A[J])

>sweat 2;
goon

[ 3 1 2]  PROVED
FAILED [ 3 1 5] (I-J =<  0)

>sweat 1;
goon

[ 3 1 5]  PROVED
FAILED [ 3 2 2] (A[ 1,J] >= A[J])

>prthm;

[(N >=  2) (K >=  1) (A[ 1,-1+J] >= A[K]) (N-J >=  0) (J-K
>=  1) (A[J]-A[K] =< -1) PERM(A[ 1,N],A0[ 1,N]) ORDERED(A[
1, 0])]
    ==>
(A[ 1,J] >= A[J])

use xf1;
goon

FAILED [ 3 2 2 -1 1] (A[ 1,-1+J] >= A[J])

>sweat 2;
goon

[ 3 2 2]  PROVED
FAILED [ 3 2 5] (J >=  1)

>sweat 1;
goon

[ 3 2 5]  PROVED
OK  [CONJ [CONJ SIMP [XF1 [ (XZ.A[J]) (XJ.J) (XI.I) (XA.A)
(XF.>=)] [CONJ [TR [ (XY.A[K])] [CONJ DEF [INST [ (XY.A[K])]
[CONJ SIMP SIMP]]]] SIMP]] SIMP SIMP [TR [ (XY.-1+J-K)] [CONJ
DEF [INST [ (XY.-1+J-K)] [CONJ SIMP SIMP]]]] SIMP SIMP SIMP]
[CONJ SIMP [XF1 [ (XZ.A[J]) (XJ.J) (XI. 1) (XA.A) (XF.>=)]
[CONJ [TR [ (XY.A[K])] [CONJ DEF [INST [ (XY.A[K])] [CONJ
SIMP SIMP]]]] SIMP]] SIMP SIMP [TR [ (XY.K)] [CONJ DEF [INST
[ (XY.K)] [CONJ SIMP SIMP]]]] SIMP SIMP SIMP]]
ASSUME FALSE

TEST ASSERTION AT L2
((I >=  1)  &  (A[I,J] >= X)  &  (N-I >=  1)  &  (N-J >=  0)
  &  (I-K =<  0)  &  (J-K >=  0)  &  (X-A[K] =  0)  &  PERM(A[
1,N],A0[ 1,N])  &  ORDERED(A[ 1,-1+I])  &  ((I =  1)  OR
(A[I,N] >= A[-1+I])))
FROM PATH CONDITION
((I >=  1)  &  (A[I,-1+J] >= X)  &  (N-I >=  1)  &  (N-J >=
 0)  &  (I-K =<  0)  &  (J-K >=  1)  &  (X-A[J] =<  0)  &
 (X-A[K] =  0)  &  PERM(A[ 1,N],A0[ 1,N])  &  ORDERED(A[ 1,-1+I])
  &  ((I =  1)  OR  (A[I,N] >= A[-1+I])))
```

```
OK  [CONJ [CONJ SIMP [XF1 [ (XZ.A[K]) (XJ.J) (XI.I) (XA.A)
(XF.>=)] [CONJ SIMP SIMP]] SIMP SIMP SIMP SIMP SIMP SIMP SIMP
SIMP] [CONJ SIMP [XF1 [ (XZ.A[K]) (XJ.J) (XI. 1) (XA.A) (XF.>=)]
[CONJ SIMP SIMP]] SIMP SIMP SIMP SIMP SIMP SIMP SIMP SIMP]]
ASSUME FALSE

AT COND  (N-I >=  2)
ASSUME TRUE

TEST ASSERTION AT L2
((I >=  0)  &  (CHNG(CHNG(A,A[I],K),X,I)[ 1+I, 1+I] >= CHNG(CHNG(
A,A[I],K),X,I)[ 1+I])  &  (N-I >=  2)  &  PERM(A0[ 1,N],CHNG(CHNG
(A,A[I],K),X,I)[ 1,N])  &  ORDERED(CHNG(CHNG(A,A[I],K),X,I)[
1,I])  &  ((I =  0)  OR  (CHNG(CHNG(A,A[I],K),X,I)[ 1+I,N]
>= CHNG(CHNG(A,A[I],K),X,I)[I])))
FROM PATH CONDITION
((I >=  1)  &  (A[I,-1+J] >= X)  &  (N-I >=  2)  &  (N-J =
-1)  &  (I-K =<  0)  &  (J-K >=  1)  &  (X-A[K] =  0)  &
PERM(A[ 1,N],A0[ 1,N])  &  ORDERED(A[ 1,-1+I])  &  ((I =
1)  OR  (A[I,N] >= A[-1+I])))
FAILED [ 5 1 2] (XCHNG(A,I,K)[ 1+I, 1+I] >= XCHNG(A,I,K)[
1+I])

>sweat 1;
goon

[ 5 1 2]  PROVED
FAILED [ 5 1 4] PERM(A0[ 1,-1+J],XCHNG(A,I,K)[ 1,-1+J])

>interm "a"<<1,"j"-1>>;

SUBGOAL PROVED SIMP
FAILED [ 5 1 4 2] PERM(A[ 1,-1+J],XCHNG(A,I,K)[ 1,-1+J])

>use perm13;
goon

FAILED [ 5 1 4 2 -1 3] (K >=  1)

>sweat 1;
goon

FAILED [ 5 1 4 2 -1 3] (K >=  1)

>sweat 1;
goon

SUBGOAL PROVED [PERM13 [ (XV.K) (XU.I) (XJ.-1+J) (XI. 1) (XA.A)]
[CONJ SIMP SIMP [TR [ (XY.-I+K)] [CONJ DEF [INST [ (XY.-I+K)]
[CONJ SIMP SIMP]]]] SIMP]]
goon

[ 5 1 4]  PROVED
FAILED [ 5 1 5] ORDERED(XCHNG(A,I,K)[ 1,I])

>assume;
```

The proof of this condition is given in Section 5.4.2.

```
FAILED [ 5 1 6] (XCHNG(A,I,K)[ 1+I,-1+J] >= XCHNG(A,I,K)[I])

>simpgoal;

(XCHNG(A,I,K)[ 1+I,-1+J] >= A[K])

use xf3;
goon

FAILED [ 5 1 6 -1] (XCHNG(A,I,K)[I,-1+J] >= A[K])

> use xchng5;
goon

[ 5 1 6]  PROVED
FAILED [ 5 2 2] (XCHNG(A, 1,K)[ 2, 2] >= XCHNG(A, 1,K)[ 2])

>sweat 1;
goon

[ 5 2 2]  PROVED
FAILED [ 5 2 4] PERM(A0[ 1,-1+J],XCHNG(A, 1,K)[ 1,-1+J])

>interm "a"<<1,"j"-1>>;

SUBGOAL PROVED SIMP
SUBGOAL PROVED [PERM13 [ (XV.K) (XU. 1) (XJ.-1+J) (XI. 1)
(XA.A)] [CONJ SIMP SIMP SIMP SIMP]]
goon

[ 5 2 4]  PROVED
FAILED [ 5 2 6] (XCHNG(A, 1,K)[ 2,-1+J] >= XCHNG(A, 1,K)[ 1])

>simpgoal;

(XCHNG(A, 1,K)[ 2,-1+J] >= A[K])

use xf3;
goon

FAILED [ 5 2 6 -1] (XCHNG(A, 1,K)[ 1,-1+J] >= A[K])

>use xchng5;
goon

[ 5 2 6]  PROVED
OK  [CONJ [CONJ SIMP [XF1 [ (XZ.XCHNG(A,I,K)[ 1+I]) (XJ. 1+I)
(XI. 1+I) (XA.XCHNG(A,I,K)) (XF.>=)] [CONJ [XF0 [ (XZ.XCHNG(A,I,K
)[ 1+I]) (XJ.I) (XI. 1+I) (XA.XCHNG(A,I,K)) (XF.>=)] SIMP]
SIMP]] SIMP [INTERM A[ 1,-1+J]] ASSUMED [XF3 [ (XX.A[K]) (XJ.-1+J
) (XI. 1+I) (XA.XCHNG(A,I,K)) (XF.>=)] [XCHNG5 [ (XZ.A[K])
(XJ.-1+J) (XI.I) (XV.K) (XU.I) (XA.A) (XF.>=)] [CONJ SIMP
SIMP SIMP SIMP SIMP]]]] [CONJ SIMP [XF1 [ (XZ.XCHNG(A, 1,K)[
2]) (XJ. 2) (XI. 2) (XA.XCHNG(A, 1,K)) (XF.>=)] [CONJ [XF0
[ (XZ.XCHNG(A, 1,K)[ 2]) (XJ. 1) (XI. 2) (XA.XCHNG(A, 1,K))
```

```
(XF.>=)] SIMP] SIMP]] SIMP [INTERM A[ 1,-1+J]] [ORD1 [ (XJ.
1) (XI. 1) (XA.XCHNG(A, 1,K))] SIMP] [XF3 [ (XX.A[K]) (XJ.-1+J)
(XI. 2) (XA.XCHNG(A, 1,K)) (XF.>=)] [XCHNG5 [ (XZ.A[K]) (XJ.-1+J)
 (XI. 1) (XV.K) (XU. 1) (XA.A) (XF.>=)] [CONJ SIMP SIMP SIMP
SIMP SIMP]]]]]
ASSUME FALSE

TEST ASSERTION AT OUT
(PERM(A0[ 1,N],CHNG(CHNG(A,A[I],K),X,I)[ 1,N]) & ORDERED(CHNG(C
HNG(A,A[I],K),X,I)[ 1,N]))
FROM PATH CONDITION
((I >= 1) & (A[I,-1+J] >= X) & (N-I = 1) & (N-J =
-1) & (I-K =< 0) & (J-K >= 1) & (X-A[K] = 0) &
PERM(A[ 1,N],A0[ 1,N]) & ORDERED(A[ 1,-1+I]) & ((I =
1) OR (A[I,N] >= A[-1+I])))
FAILED [ 6 1 1] PERM(A0[ 1,-1+J],XCHNG(A,-2+J,K)[ 1,-1+J])

>prthm;

[(J >= 3) (A[-2+J,-1+J] >= A[K]) (A[-2+J,-1+J] >= A[-3+J])
(J-K >= 1) (J-K =< 2) PERM(A[ 1,-1+J],A0[ 1,-1+J]) ORDERED(A[
1,-3+J])]
    ==>
PERM(A0[ 1,-1+J],XCHNG(A,-2+J,K)[ 1,-1+J])

interm "a"<<1,"j"-1>>;

SUBGOAL PROVED SIMP
FAILED [ 6 1 1 2] PERM(A[ 1,-1+J],XCHNG(A,-2+J,K)[ 1,-1+J])

>use perm13;
goon

FAILED [ 6 1 1 2 -1 3] (K >= 1)

>sweat 1;
goon

FAILED [ 6 1 1 2 -1 3] (K >= 1)

>interm "j"-2;

SUBGOAL PROVED SIMP
SUBGOAL PROVED SIMP
goon

SUBGOAL PROVED [PERM13 [ (XV.K) (XU.-2+J) (XJ.-1+J) (XI. 1)
(XA.A)] [CONJ SIMP SIMP [INTERM -2+J] SIMP]]
goon

[ 6 1 1]  PROVED
FAILED [ 6 1 2] ORDERED(XCHNG(A,-2+J,K)[ 1,-1+J])

>use ord12;
goon
```

```
FAILED [ 6 1 2 -1 2] (XCHNG(A,-2+J,K)[-2+J]-XCHNG(A,-2+J,K)[-1+J]
 =<  0)

>simpgoal;

(A[K]-XCHNG(A,-2+J,K)[-1+J] =<  0)

cases [% "k"="j"-2, "k"="j"-1 %];

FAILED [ 6 1 2 -1 2 1] (A[K]-XCHNG(A,K,K)[ 1+K] =<  0)

>simpgoal;

(A[K]-A[ 1+K] =<  0)

sweat 2;
goon

CASE PROVED [XF6 [ (XK. 1+K) (XI.K)] [INST [ (XK. 1+K) (XI.K)]
SIMP]]
FAILED [ 6 1 2 -1 2 2] (A[K]-XCHNG(A,-1+K,K)[K] =<  0)

>simpgoal;

(A[K]-A[-1+K] =<  0)

sweat 2;
goon

CASE PROVED [XF6 [ (XK.K) (XI.-1+K)] [INST [ (XK.K) (XI.-1+K)]
SIMP]]
CASES PROVED [(J-K =  2) (J-K =  1)]
goon

FAILED [ 6 1 2 -1 3] ORDERED(XCHNG(A,-2+J,K)[ 1,-2+J])

>cases [% 1="j"-2, 1<"j"-2 %];

CASE PROVED [ORD1 [ (XJ. 1) (XI. 1) (XA.XCHNG(A, 1,K))] SIMP]
FAILED [ 6 1 2 -1 3 2] ORDERED(XCHNG(A,-2+J,K)[ 1,-2+J])

>use ord12;
goon

FAILED [ 6 1 2 -1 3 2 -1 2] (XCHNG(A,-2+J,K)[-3+J]-XCHNG(A,-2+J,K
)[-2+J] =<  0)

>simpgoal;

(A[K]-A[-3+J] >=  0)

sweat 2;
goon

FAILED [ 6 1 2 -1 3 2 -1 3] ORDERED(XCHNG(A,-2+J,K)[ 1,-3+J])
```

```
>simpgoal;

ORDERED(A[ 1,-3+J])

goon

CASE PROVED [ORD12 [ (XJ.-2+J) (XI. 1) (XA.XCHNG(A,-2+J,K))]
[CONJ SIMP [XF6 [ (XK.-1+J) (XI.-2+J)] [INST [ (XK.-1+J) (XI.-2+J
)] [CONJ SIMP SIMP SIMP]]] SIMP]]
CASES PROVED [(J =  3) (J >=  4)]
goon
[ 6 1 2]  PROVED
FAILED [ 6 2 1] PERM(A0[ 1, 2],XCHNG(A, 1,K)[ 1, 2])

>prthm;

[(K >=  1) (K =<  2) (A[ 1, 2] >= A[K]) PERM(A[ 1, 2],A0[
1, 2]) ORDERED(A[ 1, 0])]
    ==>
PERM(A0[ 1, 2],XCHNG(A, 1,K)[ 1, 2])

interm "a"<<1,2>>;

SUBGOAL PROVED SIMP
SUBGOAL PROVED [PERM13 [ (XV.K) (XU. 1) (XJ. 2) (XI. 1) (XA.A)]
[CONJ SIMP SIMP SIMP SIMP]]
goon

[ 6 2 1]  PROVED
FAILED [ 6 2 2] ORDERED(XCHNG(A, 1,K)[ 1, 2])

>use ord2;
goon

FAILED [ 6 2 2 -1 2] (XCHNG(A, 1,K)[ 1]-XCHNG(A, 1,K)[ 2]
=<  0)

>cases [% "k"=1, "k"=2 %];

FAILED [ 6 2 2 -1 2 1] (XCHNG(A, 1, 1)[ 1]-XCHNG(A, 1, 1)[
2] =<  0)

>simpgoal;
(A[ 1]-A[ 2] =<  0)

sweat 2;
goon

CASE PROVED [XF6 [ (XK. 2) (XI. 1)] [INST [ (XK. 2) (XI. 1)]
SIMP]]
FAILED [ 6 2 2 -1 2 2] (XCHNG(A, 1, 2)[ 1]-XCHNG(A, 1, 2)[
2] =<  0)

>simpgoal;
```

```
(A[ 1]-A[ 2] >=  0)

sweat 2;
goon

CASE PROVED [XF6 [ (XK. 2) (XI. 1)] [INST [ (XK. 2) (XI. 1)]
SIMP]]
CASES PROVED [(K =  1) (K =  2)]
goon

FAILED [ 6 2 2 -1 3] ORDERED(XCHNG(A, 1,K)[ 2, 2])
>sweat 1;
goon

[ 6 2 2]  PROVED
OK  [CONJ [CONJ [INTERM A[ 1,-1+J]] [ORD12 [ (XJ.-1+J) (XI.
1) (XA.XCHNG(A,-2+J,K))] [CONJ SIMP [CASES (J-K =  2) (J-K
=  1)] [CASES (J =  3) (J >=  4)]]]] [CONJ [INTERM A[ 1, 2]]
[ORD2 [ (XJ. 2) (XI. 1) (XA.XCHNG(A, 1,K))] [CONJ SIMP [CASES
(K =  1) (K =  2)] [ORD1 [ (XJ. 2) (XI. 2) (XA.XCHNG(A, 1,K))]
SIMP]]]]]

RUN TIME = 417.022

THEOREMS PENDING:

[[(I >=  1) (A[I,-1+J] >= A[K]) (A[I,-1+J] >= A[-1+I]) (I-J
=< -3) (I-K =<  0) (J-K >=  1) PERM(A[ 1,-1+J],A0[ 1,-1+J])
ORDERED(A[ 1,-1+I])] ORDERED(XCHNG(A,I,K)[ 1,I])]
```

6.4 Derived rules

In this section we show the proof of the validity of a particular derived inference rule. The rule is DRD12 (Appendix 3), a counting-down, recursive characterization of the predicate ordered.

```
IR DRD12
  DRDERED(XA<<XI,XJ>>)
  <== [% XI<XJ, XA(XJ-1)=<XAxJ, DRDERED(XA<<XI,XJ-1>>) %];
```

We shall actually give two proofs of this rule. In the first proof we assume that ordered is defined recursively by:

```
ordered(a[i,j])  <->  i>=j or
                      a[i]=<a[i+1] & ordered(a[i+1,j]).
```

This definition is represented by the rules DRD1 to DRD4 of Appendix 2. In this case the proof is by induction on the length of the array segment. Since we are assuming that i<j we start by letting j=i+n+1 and do induction on n.

In the second proof we define ordered in the more direct way by:

```
ordered(a[i,j])  <->  (FA x)(i=<x & x<j => a[x]=<a[x+1]),
```

using the two inference rules

```
IR NORD1
  DRDERED(XA<<XI,XJ>>)
  <== [% (FA XX)(XI=<XX & XX<XJ =>> XA\XX=<XA\(XX+1)) %];

IR NORD2
  XA\XX=<XA\(XX+1)
  <== [% (EX [XI XJ])(XI=<XX & XX<XJ &
                      DRDERED(XA<<XI,XJ>>)) %];
```

In this case the proof reduces to a case analysis. In practice, given this definition the rule DRD12 would not also be required but the proof is indicative of the way the

rules NORD1 and NORD2 are used.

In each case, the theorem to be proved is

i<j & a[j-1]=<a[j] & ordered(a[i,j-1]) => ordered(a[i,j])

First proof

```
ORD   14-5-1975
MAXDEPTH= 2  CONDPROVE= 0  TRACE= 0
FAILED [ 3] ORDERED(XA[XI,XJ])

>prthm;

[(XI-XJ =< -1) (XA[XJ]-XA[-1+XJ] >=  0) ORDERED(XA[XI,-1+XJ])]
    ==>
ORDERED(XA[XI,XJ])

add xj=xi+xn+1;
subst(xi+xn+1,xj,goal) -> goal;
subst(xi+xn+1,xj,hyps) -> hyps;
```

This substitutes i+n+1 for j throughout the theorem.

```
prthm;

[(XN >=  0) (XA[ 1+XI+XN]-XA[XI+XN] >=  0) ORDERED(XA[XI,XI+XN])]

    ==>
ORDERED(XA[XI, 1+XI+XN])

induct xn;

BASIS PROVED [ORD2 [ (XJ. 1+XI) (XI.XI) (XA.XA)] [CONJ SIMP
SIMP [ORD1 [ (XJ. 1+XI) (XI. 1+XI) (XA.XA)] SIMP]]]
FAILED [ 3 2] ORDERED(XA[XI, 2+XI+XN01])

>use ord2;
goon

FAILED [ 3 2 -1 2] (XA[XI]-XA[ 1+XI] =<  0)

>use ord3;
goon

FAILED [ 3 2 -1 3] ORDERED(XA[ 1+XI, 2+XI+XN01])
```

The induction hypothesis is the rule
ordered(xa[xi,xi+xn01+1]) <- ...
The two conclusions will not match unless the instantiations for the variables of the induction hypothesis are given explicitly. In doing this, the first xi is the one in the rule and the second is the one in the goal.

```
>let xi be xi+1 in indhyp;
goon

STEP PROVED [ORD2 [ (XI. 1+XI)] [CONJ SIMP [ORD3 [ (XJ. 1+XI+XN01
) (XI.XI) (XA.XA)] [INST [ (XJ. 1+XI+XN01) (XI.XI) (XA.XA)]
[CONJ SIMP SIMP]]] [INDHYP [ (XI.XI) (XA.XA)] [CONJ SIMP SIMP
[ORD4 [ (XJ. 1+XI+XN01) (XI. 1+XI) (XA.XA)] [CONJ SIMP SIMP]]]]]]

goon

[ 3]  PROVED
```

Second proof

```
ORD   14-5-1975
MAXDEPTH= 2  CONDPROVE= 0  TRACE= 0

FAILED [ 4] ORDERED(XA[XI,XJ])

>prthm;

[(XI-XJ =< -1) (XA[XJ]-XA[-1+XJ] >=  0) ORDERED(XA[XI,-1+XJ])]
    ==>
ORDERED(XA[XI,XJ])

use nord1;
goon

FAILED [ 4 -1] (XA[XX02]-XA[ 1+XX02] =<  0)

          XX02 is a Skolem constant introduced from  the  FA
          expression.

>prthm;

[(XI-XJ =< -1) (XI-XX02 =<  0) (XJ-XX02 >=  1) (XA[XJ]-XA[-1+XJ]
>=  0) ORDERED(XA[XI,-1+XJ])]
    ==>
(XA[XX02]-XA[ 1+XX02] =<  0)

cases [% xx02=xj-1, xx02<xj-1 %];

CASE PROVED SIMP
FAILED [ 4 -1 2] (XA[XX02]-XA[ 1+XX02] =<  0)

>use nord2;
goon

CASE PROVED [NORD2 [ (XJ.-1+XJ) (XI.XI) (XA.XA)] [INST [ (XJ.-1+X
J) (XI.XI) (XA.XA)] [CONJ SIMP SIMP SIMP]]]
CASES PROVED [(XJ-XX02 =  1) (XJ-XX02 >=  2)]
goon

[ 4]  PROVED
```

Chapter 7. CONCLUSIONS

7.1 Summary

We have presented in this thesis a new method for proving properties of programs and an implemented verification system which uses both this method and Floyd's method. The proof method is based on symbolic execution and as such relies on the operational semantics of a language. It is a generalization of Floyd's method and can be effectively used in conjunction with it, especially for handling subroutines and recursive procedures. Moreover, it can also handle language features such as non-local jumps, side-effects and non-determinism. While the method is very convenient to use for some programs (e.g. the 91-function), for others it can require more detailed specifications than are really necessary and hence makes these programs harder to verify (e.g. GCD, Section 3.3.1). Thus the method is perhaps best seen as a complement to existing methods, one which is occasionally preferable to any of them.

The program verification system is interesting for several reasons: firstly for the way it "borrows" the semantics of the language as actually implemented, thereby obviating the need for a formal definition of the language, secondly for the use of high-level specifications, and thirdly for the way an interactive theorem prover can augment a simple automatic one, enabling non-trivial programs, including two sorting programs and a program to

invert a permutation "in place", to be verified. It is also an extensible system, allowing new functions to be declared, defined by rules, and given properties which are then used by the simplifier and matcher. The major weakness of the verification system is that, with the exception of the permutation example, we have not been able to use it to verify any difficult programs which have not already been verified completely automatically by other systems. Even those proofs we have done have been quite hard to find (certainly while sitting at a terminal) and tedious to check. Finally, the system can be very slow, especially considering that its automatic theorem prover possesses almost no "intelligence". Nevertheless the system has achieved a limited success in a difficult area, and with the modifications and extensions described in the next section could be a useful tool.

7.2 Improvements and extensions

Several factors contribute to the deficiencies just mentioned. These include the lack of a routine for simplifying conjunctions of more than two relational expressions (e.g. our system cannot simplify a>b & b>c & c>a to false), the implementation of transitivity as an inference rule requiring the use of the matcher, the fact that rules are applied without considering the hypotheses and thus leading to a rather blind search, the limited knowledge about quantifiers and sets, and the excessive use

of the algebraic simplifier.

Some of these difficulties can be overcome in straightforward ways. For example, using a unique representation for identical expressions (as in QA4) allows a property list to be associated with each expression which can facilitate transitivity and set operations, and avoid repeated computations of related expressions (e.g. negations). Alternatively, and more simply in our system, adding the typing and cases mechanism described briefly in Good et al. (1974) would also improve our ability to reason about relational expressions.

A more difficult problem is deciding which inference rule to apply at any given point. It is clearly necessary to consider the hypotheses of both the theorem and the rule when doing this. A generalization of our current indexing scheme which only applied a rule automatically if every "interesting" predicate in the rule's hypotheses also occurred in the theorem's hypotheses would surrender completeness (which the automatic theorem prover doesn't have anyway), but it would drastically reduce the search space, allow the search to go deeper and possibly result in improved performance.

But the most important problem is the nature of the rules themselves. There are too many rules doing similiar things, the user needs to know exactly what they each are, and must be able to decide which one is currently applicable. More descriptive names alone would help, but a

better solution is to give "natural" rather than recursive definitions of new functions, for example,

ordered(a[x,y]) <-> (FA u)(x=<u<y => a[u]=<a[u+1]) .

Our various counting-up and counting-down rules would then be combined into rules about the quantifiers only, and to prove ordered(a[i,j]) the user would say "count-down" rather than "use ORD12". This would also allow uniform heuristics for quantifiers to be introduced, thereby lessening still further the burden on the user. We are currently modifying the system to use such non-recursive definitions.

Extensions which would make the system more useful include the ability to do proofs by Burstall's hand simulation method, and the provision of routines for manipulating explicit sets, including sets defined by abstraction. The first of these would allow the proofs of programs such as the Schorr-Waite program to be attempted, and the second would be an additional aid to writing assertions in the most natural way.

7.3 Further research

There are two dimensions in program verification: towards increased program complexity, and toward increased programming language complexity. This thesis has considered both aspects, in the verifier and the proof method respectively. While it is a non-trivial task to devise a

proof system and write a verification condition generator for a language with powerful features (as evidenced by the present work and by Igarashi, London and Luckham, 1973), we now believe that this is much the less important direction of the two. Even in the simplest language it is possible to write arbitrarily complicated programs which require a considerable body of extra-programming knowledge in order to be verified. Thus it seems that mechanized program verifiers will only be as successful as the theorem provers / proof checkers which they use.

The development of effective, powerful interactive theorem provers for use in program verification remains a difficult problem. There is the question of what is the best way to organize such an interactive theorem prover. The approach taken by most workers in program verification, including ourselves, is to supplement an automatic theorem prover by commands from the user. However, in LCF, the basic program is a low-level proof checker controlled by the user, to which successively more powerful derived inference rules are added. There is also the choice between an LCF / Bledsoe natural deduction system and a King / Deutsch / resolution system. In the first case it probably does not make much difference which choice is taken since they will come to the same thing in the end. However it seems that interactive systems should be based on a natural deduction approach rather than one based on deriving contradictions, simply because the resulting proofs in a natural deduction system are more similar to the ones people produce, and

hence people can more easily interact with and guide such a system.

For program verification to become a widespread, profitable activity, it will also be necessary to remove the burden of giving detailed specifications from the user. Some work has been done on automatically generating loop invariants from input-output assertions (German and Wegbreit 1975), but this by itself seems to be an artificial problem, especially since it requires the program to be correct in the first place. Moreover, as German and Wegbreit observe, since the programmer presumably knows why his program works, he can always give at least an outline of this loop invariant. It should then be practical to complete this invariant, adapting the techniques of Sites (1974) to find array subscript bounds, etc., and use it to verify the program.

It might also be possible, especially when using continuation induction, to adapt the ideas of Moore (1974) and Aubin (1975) who have independently found methods for extending the generalization techniques of the Boyer-Moore LISP theorem prover to handle iterative programs. Both methods involve the introduction of recursive functions describing the state of the computation after an arbitrary number of iterations. While these new functions do not exactly describe the computation to the end of the loops, the fact that in both cases induction is done on the remaining length of the computation suggests that their

methods could be applied in deriving the function computed by a particular loop, particularly if a partial description was given initially.

To carry these ideas even further, the most fruitful approach could well be that of interactively constructing the assertions and the program simultaneously, continually testing their consistency, and using the results of these tests in their further development. This approach was advocated in Floyd (1971) and was shown to be feasible by Deutsch.

Finally, to verify large programs it will be necessary to adopt the principles of abstraction and structure as advocated in Dahl, Dijkstra and Hoare (1972) for example. As Good, London and Bledsoe observe, to keep the size of proofs manageable, abstractions must be used in the specifications as well as in the programs. Some progress has been made in this direction by the use of array segments and defined predicates (ordered, perm) but most programs which have been mechanically verified are still too simple for the advantages of this approach to be manifest. The chief exception is the verification condition generator proved correct by Ragland (1973). The next step is to start by verifying the truly abstract programs which operate on sets, bags, graphs, etc. and then showing that the concrete program correctly simulates the abstract program using, for example, the method described in Hoare (1972).

The future progress of (interactive) program verification is unclear. There are now several well-understood and practical methods for proving properties of programs, and it is a question of implementing them as effectively as possible. But despite a considerable amount of research in the last three years, and the implementation of several new verification systems, the complexity of verified programs has only increased modestly. To verify more complex programs it will be necessary to whole-heartedly adopt the structuring and abstraction principles referred to above, thereby keeping the proofs manageable. There is no reason this cannot be done, and with some effort the field should continue to make steady progress.

REFERENCES

Aubin, R. (1975) Some generalization heuristics in proofs by induction. Proceedings of the International Symposium on Proving and Improving Programs, Arc-et-Senans, France.

de Bakker, J.W. and Scott, D. (1969) A theory of programs. Unpublished memo, Vienna.

Basu, S.K. and Misra, J. (1975) Proving loop programs. IEEE Transactions on Software Engineering 1, 1, 76-86.

Bledsoe, W.W. and Bruell, P. (1973) A man-machine theorem proving system. Proceedings of Third International Joint Conference on Artificial Intelligence, Stanford, California, pp. 56-65. Also Artificial Intelligence Journal 5, 1 (1974) 51-72.

Boyer, R., Elspas, B. and Levitt, K.N. (1975) SELECT - a formal system for testing and debugging programs by symbolic execution. Proceedings of the International Conference on Reliable Software, Los Angeles, California, pp. 234-245.

Boyer, R. and Moore J S. (1973) Proving theorems about LISP functions. Proceedings of Third International Joint Conference on Artificial Intelligence, Stanford, California, pp. 486-493. Also J. ACM 22, 1 (1975) 129-144.

Burstall, R.M. (1969) Proving properties of programs by structural induction. Computer Journal 12, 1, 41-47.

Burstall, R.M., Collins, J.S. and Popplestone, R.J. (1971) Programming in POP-2. Edinburgh: University Press.

Burstall, R.M. (1974) Program proving as hand simulation with a little induction. Proceedings of IFIP Congress 1974, Stockholm, Sweden.

Burstall, R.M. (1975) A note on program proof by a continuation method. DAI Working Paper 2. Department of Artificial Intelligence, University of Edinburgh.

Clint, M. and Hoare, C.A.R. (1972) Program proving: jumps and functions. Acta Informatica 1, 214-224.

Cooper, D.C. (1971) Programs for mechanical program verification. Machine Intelligence 6 (eds. B. Meltzer and D. Michie) Edinburgh: University Press, pp. 43-59.

Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R. (1972) Structured Programming. Academic Press, London.

Darlington, J. (1973) A semantic approach to automatic program improvement, Ph.D. thesis, University of Edinburgh.

Deutsch, L.P. (1973) An interactive program verifier. Ph.D. thesis. University of California, Berkeley, California. Also Xerox PARC Report CSL-73-1, Palto Alto, California.

Elspas, B. (1972) The semiautomatic generation of inductive assertions for program correctness proofs. Report No. 55, Seminar, Des Institute fur Theorie der Automaten und Schaltnetzwerke, Gesellschaft fur Mathematik und Datenverarbeitung, Bonn. Also SRI Project 2686, Stanford Research Institute, Menlo Park, California (1974).

Elspas, B., Levitt, K.N., Waldinger, R.J. and Waksman, A. (1972) An assessment of techniques for proving program correctness. ACM Computing Surveys 4, 2, 97-147.

Floyd, R.W. (1964) Algorithm 245, TREESORT 3. Comm. ACM 7, 12, 701.

Floyd, R.W. (1967) Assigning meanings to programs. Proceedings of a Symposium in Applied Mathematics, Vol. 19 (ed. J.T. Schwartz), Providence, Rhode Island, American Mathematical Society, pp. 19-32.

Floyd, R.W. (1971) Toward interactive design of correct programs. Proceedings of IFIP Congress 1971, Amsterdam, Netherlands.

Gerhart, S.L. (1972) Verification of APL programs. Ph.D. thesis. Carnegie-Mellon University, Pittsburgh, Pennsylvania.

German, S.M. and Wegbreit, B. (1975) A synthesizer of inductive assertions. IEEE Transactions on Software Engineering 1, 1, 68-75.

Good, D.I. (1970) Toward a man-machine system for proving program correctness. Ph.D. thesis. University of Wisconsin.

Good, D.I., London, R.L. and Bledsoe, W.W. (1974) An interactive program verification system. ISI/RR-74-22. Information Sciences Institute, University of Southern California. Also IEEE Transactions on Software Engineering 1, 1 (1975) 59-67.

von Henke, F.W. and Luckham, D.C. (1974) Automatic program verification III: A methodology for verifying programs. Computer Science Department Report CS-474, AIM-256, Stanford University, California. Also Proceedings of International Conference on Reliable Software, Los Angeles, California (1975) pp.156-164.

Hoare, C.A.R. (1969) An axiomatic basis for computer programming. Comm. ACM 12, 10, 576-580, 583.

Hoare, C.A.R. (1971a) Procedures and parameters: an axiomatic approach. Symposium on the Semantics of Algorithmic Languages (ed. E. Engeler). Lecture Notes in Mathematics 188. Springer-Verlag, pp. 102-116.

Hoare, C.A.R. (1971b) Proof of a program: FIND. Comm. ACM 14, 1, 39-45.

Hoare, C.A.R. (1972) Proof of correctness of data representations. Acta Informatica 1, 271-278.

Igarashi, S., London, R. and Luckham, D. (1973) Automatic verification of programs I: a logical basis and implementation. Computer Science Department Report CS-365, AIM-200, Stanford University, California.

Katz, S.M. and Manna, Z. (1973) A heuristic approach to program verification. Proceedings of Third International Joint Conference on Artificial Intelligence, Stanford, California, pp. 500-512.

King, J.C. (1969) A program verifier. Ph.D. thesis. Carnegie-Mellon University, Pittsburgh, Pennsylvania.

King, J.C. (1975) A new approach to program testing. Proceedings of the International Conference on Reliable Software, Los Angeles, California, pp. 228-233.

Knuth, D.E. (1968) The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass.

Landin, P.J. (1964) The mechanical evaluation of expressions. Computer Journal 6, 308-320.

Leisenring, A.C. (1969) Mathematical Logic and Hilbert's ε-Symbol. MacDonald and Co., London.

Manna, Z. (1969) The correctness of programs. Journal of Computer and Systems Sciences 3, 119-127.

Manna, Z. and Pnueli, A. (1970) Formalization of properties of functional programs. J. ACM 12, 3, 555-569.

Manna, Z., Ness, S. and Vuillemin, J. (1971) Inductive methods for proving properties of programs. Proceedings of the ACM SIGPLAN Conference on Proving Assertions about Programs, Las Cruces, New Mexico, pp. 27-50. Also Comm. ACM 16, 8 (1973) 491-502.

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM 3, 184-195.

McCarthy, J. (1962) Toward a mathematical science of computation. Information Processing 1962 (ed. C.M. Popplewell). Proceedings of IFIP Congress 1962. Amsterdam: North-Holland.

McCarthy, J. (1963) A basis for a mathematical theory of computation. Computer Programming and Formal Systems (eds. P. Braffort and D. Hirschberg) Amsterdam: North-Holland, pp. 33-70.

McCarthy, J. and Painter, J.A. (1967) Correctness of a compiler for arithmetic expressions. Proceedings of a Symposium in Applied Mathematics, Vol. 19 (ed. J.T. Schwartz), Providence, Rhode Island, American Mathematical Society, pp. 33-41.

Mills, H.D. (1975) The new math of computer programming. Comm. ACM 18, 1, 43-48.

Milner, R. (1972) Implementation and applications of Scott's logic for computable functions. Proceedings of the ACM SIGPLAN Conference on Proving Assertions About Programs, Las Cruces, New Mexico, pp. 1-6.

Milner, R. (1975) Private communication.

Milner, R. and Weyhrauch, R. (1972) Proving compiler correctness in a mechanised logic. Machine Intelligence 7 (eds. B. Meltzer and D. Michie) Edinburgh: University Press, pp. 51-70.

Moore, J S. (1973) Computational logic: Structure sharing and proving program properties. Ph.D. thesis, University of Edinburgh.

Moore, J S. (1974) Introducing iteration into the pure LISP theorem prover. Xerox PARC Report CSL-74-3, Palo Alto, California.

Morris, J.H. (1971) Another recursion induction principle. Comm. ACM 14, 5, 351-354.

Moses, J. (1971) Algebraic simplification: a guide for the perplexed. Comm. ACM 14, 8, 527-537.

Naur, P. (1966) Proof of algorithms by general snapshots. BIT 6, 310-316.

Newey, M.C. (1975) Formal semantics of LISP with applications to program correctness. Computer Science Department Report CS-475, AIM-252, Stanford University, California.

Park, D. (1969) Fixpoint induction and proofs of program properties. Machine Intelligence 5 (eds. B. Meltzer and D. Michie) Edinburgh: University Press, pp. 59-78.

Perlis, A.J. and Itturiaga, R. (1964) An extension to ALGOL for manipulating formula. Comm. ACM 7, 2, 127-130.

Ragland, L.C. (1973) A verified program verifier. Technical Report No. 18, Department of Computer Sciences, University of Texas at Austin.

Reynolds, J.C. (1972) Definitional interpreters for higher-order programming languages. Proceedings of ACM National Conference, Boston, Mass.

Rulifson, J.F., Derkson, J.A. and Waldinger, R.J. (1972) QA4: a procedural calculus for intuitive reasoning. Technical Note 73, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California.

Scott, D. (1969) A type theoretical alternative to ISWIM, CUCH, DWHY. Unpublished notes, Oxford University.

Sites, R.L. (1974) Clean termination of programs. Computer Science Department Report CS-418, Stanford University, California,

Stansfield, J.L. (1972) [PROCESS 1]: A generalisation of recursive programming languages. Bionics Research Reports: No. 8, School of Artificial Intelligence, University of Edinburgh.

Suzuki, N. (1974) Automatic program verification II: Verifying programs by algebraic and logical reduction. Computer Science Department Report CS-423, AIM-255, Stanford University, California. Also Proceedings of International Conference on Reliable Software, Los Angeles, California (1975) pp. 473-481.

Topor, R.W. (1974) The correctness of the Schorr-Waite list marking algorithm. MIP-R-104, School of Artificial Intelligence, University of Edinburgh.

Topor, R.W. and Burstall, R.M. (Feb. 1973) Proving programs by symbolic execution. Working note 1, University of Edinburgh. (Privately circulated)

Waldinger, R.J. and Levitt, K.N. (1973) Reasoning about programs. ACM Symposium on Principles of Programming Languages, Boston, Mass., pp. 169-182. Also Artificial Intelligence Journal 5, 3 (1974) 235-316.

Wegbreit, B. (1973) Heuristic methods for mechanically deriving inductive assertions. Proceedings of Third International Joint Conference on Artificial Intelligence, Stanford, California, pp. 524-536. Also Comm. ACM 17, 2 (1974) 102-112.

Weir, S. and Burstall, R.M. (1972) Macro-inference steps in proofs of program correctness. MIP-R-96, School of Artificial Intelligence, University of Edinburgh.

Wirth, N. (1971) The programming language Pascal. Acta Informatica 1, 35-63.

Appendix 1: Notes on POP-2

In this appendix we give a brief account of those aspects of the POP-2 programming language neccessary for the reader to be able to follow our examples and description. The language was designed and implemented at the University of Edinburgh and is described in detail in Burstall, Collins and Popplestone (1971).

POP-2 allows a programmer to represent and manipulate various kinds of objects including numbers, words, arrays, lists, strings, records and functions. There is no distinction between functions and procedures. All assignments, argument passing and result returning is done using a pushdown stack which is freely accessible to the programmer. To push the number 6 onto the stack and leave it there, one writes

```
6;
```

If there is something on the stack, it can be popped and assigned to the variable X by writing

```
-> X;
```

Thus to assign 6 to X one writes

```
6 -> X;                    (cf.  X := 6; in ALGOL)
```

One can also use the stack to interchange the values of two variables X and Y by writing

```
X, Y -> X -> Y;
```

Function application is invoked by writing, for example,

```
F(1,2);
```

The sequence of actions in this application is that the arguments 1 and 2 are placed on the stack (after having been evaluated) and the function stored in the identifier F is then applied. F then takes its arguments off the stack. Thus the same result is obtained by writing

```
1, F(2);
```

or

```
1, 2; F();
```

Function applications can also be signified by using a dot (".") before any identifier (or expression) whose value is a function. So the above example can also be written as

```
1, 2 .F;
```

Any function may return more than one result, simply by leaving the results on the stack when it exits.

To define a function with name FOO, formal parameters X and Y, and local variables Z and W, one writes

```
FUNCTION FOO X Y;
VARS Z W;
  body
END
```

For example, the function definition

```
FUNCTION SUMSQ X Y;
  X^2 + Y^2
ENO
```

defines a new function called SUMSQ whose value is the sum of the squares of its two arguments. It is possible to declare local variables so that their values are

automatically placed on the stack when the execution of the function is terminated. Such variables are called output locals. The function SUMSQ could also be defined using output locals as

```
FUNCTION SUMSQ X Y => Z;
  X^2 + Y^2 -> Z
END
```

In this case Z is the output local and its value is placed on the stack immediately before the function exits. Anonymous functions, or lambda expressions, may also be constructed and assigned to variables. For example,

```
LAMBDA X Y; X^3 + Y^3 ENO -> F;
```

assigns the anonymous function which computes the sum of the cubes of its two arguments to the variable F which can then be used like any other function.

Non-local variables in POP-2 take their values from their dynamic environment as in LISP rather than from their textual (or static) environment as in ALGOL.

The conditional statement in POP-2 has the form

```
IF cond THEN stat1 ELSE stat2 CLOSE;
```

If the condition is true (i.e. it does not evaluate to 0) then the first alternative statement is executed; otherwise the second statement is executed. FALSE in POP-2 is 0 and TRUE is 1. The form

```
IF cond1 THEN stat1
ELSEIF cond2 THEN stat2
      .....
ELSE statn CLOSE;
```

can be used to avoid repeated CLOSE's.

The main iterative statement in POP-2 has the form

```
LOOPIF cond1 THEN stat1
ELSEIF cond2 THEN stat2
      .....
ELSEIF condn THEN statn CLOSE;
```

This statement evaluates each condition in turn until one of them is true (i.e. not 0). It then executes the corresponding sub-statement and repeats the whole statement. If each condition evaluates to 0 the statement terminates. (This statement is a deterministic version of Dijkstra's guarded command construction.) The simple case

```
LOOPIF cond THEN stat CLOSE
```

corresponds to the familiar while-statement:

```
while cond do stat
```

of other languages. Labels and GOTO's have roughly the same syntax as in ALGOL.

However, since a GOTO statement cannot refer to a label outside the function body in which it occurs, the standard function JUMPOUT is provided to construct escape functions. We write, for example,

```
FUNCTION F X;
  IF X=0 THEN ERROR() CLOSE;
  (X+1) / X
END;

FUNCTION G Y;
  JUMPOUT(LAMBDA; PR('ZERO ERROR') END, 0) ->
 ERROR;
  F(Y) + F(Y^3)
END
```

After JUMPOUT in this example, ERROR is a function of no arguments and no results. In fact it is identical to the function LAMBDA; PR('ZERO ERROR') END except that as soon as ERROR has been executed, execution of G is terminated

instead of execution of F being resumed as one would normally expect. That is, ERROR has a special "fire-escape" which enables it to climb out of G (the function where it was created) when it is called. The second argument, 0, of JUMPOUT indicates that the function produced returns no results.

A more general state-saving facility allows the user to save the complete state of his computation below a "barrier", enabling more complex control structures such as backtracking and coroutines to be implemented.

Data structures in POP-2 include words, lists, arrays and records. Lists are basically the same as in LISP. There are several ways to construct them. For example, the expressions

```
[A B C D]
[% "A", "B", "C", "D" %]
CONS("A",[B C D])
"A" :: [B C D]
[A B] <> [C D]
```

all have the same value, a list consisting of the four words "A", "B", "C" and "D". The binary operation :: is the same as the function CONS and the binary operation <> is the same as the LISP function APPEND. The other main standard functions which operate on lists are HD (cf. CAR) which returns the first element of a list, TL (cf. CDR) which returns all but the first element of the list, and NULL which tests whether or not the list is equal to NIL.

Arrays in POP-2 are just particular kinds of functions; A(I) is the Ith element of the array A. (In our verification system, this is written A\I.) To assign 0 to the first element of an array A one writes

```
0 -> A(1);
```

This leads to the concept of a <u>doublet</u>. Every function can be given an associated function called an <u>updater</u>. A function which has an updater is called a doublet. The function itself is applied normally but the updater is applied when it is the top-level function to the right of the assignment arrow ("->"). Both HD and TL have updaters, so after performing

```
[A B C] -> X;
1 -> HD(X);              (cf. (RPLACA X 1) in LISP)
```

the value of X is the list [1 B C]. Another standard doublet is MEANING which is used to associate information with words and hence can be used to implement property lists in POP-2.

Pairs are an example of records having two components, *FRONT and BACK which are both doublets.* A pair is formed with the function CONSPAIR which takes two arguments and constructs a record with two components. List cells are actually pairs except that the TL of a list in POP-2 must be NIL or another list, while the back of a pair can be anything. The standard function ATOM returns TRUE if its argument is not a pair and FALSE if it is.

Appendix 2: Definitions

```
COMMENT ARITH;

WHEN [% XU>=0 %], ABS(XU) ==> XU;
WHEN [% XU<0 %],  ABS(XU) ==> MINUS(XU);

WHEN [% XU>=0 %], SGN(XU) ==> 1;
WHEN [% XU<0 %],  SGN(XU) ==> MINUS(1);

IR TR;     COMMENT TRANSITIVITY AXIOM;
  XF(XX,XZ) <== [% TRANSOP("XF"),
                   (EX XY)(XF(XX,XY) & XF(XY,XZ))  %];
TORULES;

COMMENT TORULES INDEXES THE PRECEDING RULE UNDER =, >= and =<;

IR XSQ;
  XX^2 >= 0  <==  NIL;

COMMENT DIV AND REM;

WHEN [% 0=<XU, XU<XV %],  XU DIV XV ==> 0;
WHEN [% XU>=XV, XV>0 %],  XU DIV XV ==> 1 + (XU-XV) DIV XV;

WHEN [% 0=<XU, XU<XV %],  XU REM XV ==> XU;
WHEN [% XU>=XV, XV>0 %],  XU REM XV ==> (XU-XV) REM XV;

COMMENT FAC;

WHEN [% XU=0 %],  FACTORIAL(XU) ==> 1;
WHEN [% XU>0 %],  FACTORIAL(XU) ==> XU*FACTORIAL(XU-1);

COMMENT GCD;

WHEN [% XU=0 %],  GCD(XU,XV) ==> ABS(XV);
WHEN [% XV=0 %],  GCD(XU,XV) ==> ABS(XU);
COMMENT SINCE GCD IS COMMUTATIVE ONLY ONE OF THESE RULES
         EXISTS IN THE ACTUAL SYSTEM;
WHEN [% XX/=0 %],  GCD(XX,XX) ==> XX;
WHEN [% XU/=0, XV/=0 %],  GCD(XU,XV) ==> GCD(XV, XU REM XV);

COMMENT CHNG (AND XCHNG);

WHEN [% XJ=XI %],  CHNG(XA,XX,XI)\XJ ==> XX;
WHEN [% XJ/=XI %], CHNG(XA,XX,XI)\XJ ==> XA\XJ;

WHEN [% XK=XI %], XCHNG(XA,XI,XJ)\XK ==> XA\XJ;
WHEN [% XK=XJ %], XCHNG(XA,XI,XJ)\XK ==> XA\XI;
WHEN [% XK/=XI, XK/=XJ %], XCHNG(XA,XI,XJ)\XK ==> XA\XK;

COMMENT SETS;
```

```
IR XF0
  XF(XA<<XI,XJ>>,XZ)  <==  [% XI>XJ %];
TORULES;
IR XF1;
  XF(XA<<XI,XJ>>,XZ)
  <== [% XF(XA<<XI,XJ-1>>,XZ), XF(XA\XJ,XZ) %];
TORULES;
IR XF2;
  XF(XA<<XI,XJ>>,XZ)
  <== [% XF(XA<<XI+1,XJ>>,XZ), XF(XA\XI,XZ) %];
TORULES;
IR XF3
  XF(XA<<XI,XJ>>,XX)  <==  [% XF(XA<<XI-1,XJ>>,XX) %];
TORULES;
IR XF4
  XF(XA<<XI,XJ>>,XX)  <==  [% XF(XA<<XI,XJ+1>>,XX) %];
TORULES;
IR XF5
  XF(XX-XA\XJ,0)
  <==  [% (EX [XI XK])(XI=<XJ & XJ=<XK & XF(XX, XA<<XI,XK>>)) %];
TORULES;
IR XF6
  XF(XA\XJ-XX,0)
  <==  [% (EX [XI XK])(XI=<XJ & XJ=<XK & XF(XA<<XI,XK>>, XX)) %];
TORULES;

COMMENT EQSEQ;

IR EQSEQ1
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>) <== [% XI>XJ, XK>XM %];
IR EQSEQ2
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% XI=<XJ, XJ-XI=XM-XK, XA\XI=XB\XK,
         EQSEQ(XA<<XI+1,XJ>>,XB<<XK+1,XM>>) %];
IR EQSEQ3
  XA\XI=XB\XK
  <== [% (EX [XJ XM])(XI=<XJ & XM-XK=XJ-XI &
                      EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>)) %];
IR EQSEQ4
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% XI-1=<XJ, XM-XK=XJ-XI,
 EQSEQ(XA<<XI-1,XJ>>,XB<<XK-1,XM>>) %];

COMMENT ISIN;

IR ISIN1
  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% XJ-XI=<XM-XK, EQSEQ(XA<<XI,XJ>>,XB<<XK,XK+XJ-XI>>) %];
IR ISIN2
  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% ISIN(XA<<XI,XJ>>,XB<<XK+1,XM>>) %];
IR ISIN3
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XL>>)
  <== [% (EX XM)(XL=<XM & XJ-XI=<XM-XK &
                  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)) %];
```

```
IR ISIN4
  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)
  <==  [% XJ-XI=<XM-XK+1, ISIN(XA<<XI,XJ>>,XB<<XK-1,XM>>),
          NOT(EQSEQ(XA<<XI,XJ>>,XB<<XK-1,XK-1+XJ-XI>>))  %];

COMMENT ORD(EREL);

IR ORD1
  ORDERED(XA<<XI,XJ>>)  <==  [% XI>=XJ %];
IR ORD2
  ORDERED(XA<<XI,XJ>>)
  <==  [% XI<XJ, XA\XI=<XA\(XI+1), ORDERED(XA<<XI+1,XJ>>) %];
IR ORD3
  XA\XI =< XA\(XI+1)
  <==  [% (EX XJ)(XI<XJ & ORDERED(XA<<XI,XJ>>)) %];
IR ORD4
  ORDERED(XA<<XI,XJ>>)
  <==  [% XI-1<XJ, ORDERED(XA<<XI-1,XJ>>) %];

COMMENT PERM;

WHEN [% XJ<XI %],  OCCS(XX,XA<<XI,XJ>>) ==> 0;
WHEN [% XI=<XJ, XX=XA\XI %],
     OCCS(XX,XA<<XI,XJ>>) ==> 1+OCCS(XX,XA<<XI+1,XJ>>);
WHEN [% XI=<XJ, XX/=XA\XI %],
     OCCS(XX,XA<<XI,XJ>>) ==> OCCS(XX,XA<<XI+1,XJ>>);

IR PERM1
  PERM(XA<<XI,XJ>>,XB<<XK,XM>>)
  <==  [% (FA XX)(OCCS(XX,XA<<XI,XJ>>)=OCCS(XX,XB<<XK,XM>>)) %];
IR PERM2
  OCCS(XX,XA<<XI,XJ>>) = OCCS(XX,XB<<XK,XM>>)
  <==  [% PERM(XA<<XI,XJ>>,XB<<XK,XM>>) %];

COMMENT LISTS (AND TREES);

HD(CONS(XU,XV)) ==> XU;
TL(CONS(XU,XV)) ==> XV;
WHEN [% NOT(NULL(XU)) %], CONS(HD(XU),TL(XU)) ==> XU;

WHEN [% NULL(XU) %], XU<>XV ==> XV;
WHEN [% NOT(NULL(XU)) %], XU<>XV ==> HD(XU)::(TL(XU)<>XV);

IR MEMB1
  MEMB(XU,XL) <== [% NOT(NULL(XL)), XU=HD(XL) %];
IR MEMB2
  MEMB(XU,XL) <== [% NOT(NULL(XL)), MEMB(XU,TL(XL)) %];
IR MEMB3
  NOT(NULL(XL)) <== [% (EX "XU")(MEMB(XU,XL)) %];
IR MEMB4
  MEMB(XU,TL(XL)) <== [% XU/=HD(XL), MEMB(XU,XL) %];

WHEN [% ATOM(XL) %], FRINGE(XL) ==> XL::NIL;
WHEN [% NOT(ATOM(XL)) %],
     FRINGE(XL) ==> FRINGE(FRONT(XL))<>FRINGE(BACK(XL));
```

```
NULL(NIL) ==> TRUE;
NULL(CONS(XU,XV)) ==> FALSE;
ATOM(CONS(XU,XV)) ==> FALSE;
```

Appendix 3: Derived rules

```
WHEN [% XX>=0, XY>0, XZ>=0 %],
     (XX+XY*XZ) DIV XY ==> XX DIV XY + XZ;
WHEN [% XX>=0, XY>0, XZ>=0 %],
     (XX+XY*XZ) REM XY ==> XX REM XY;
XX DIV 1 ==> XX;
(XX*XY) DIV (XX*XZ) ==> XY DIV XZ;
WHEN [% XX REM XY = 0 %], (XX DIV XY)*XY ==> XX;

IR DIV1;
  XX DIV XY >=0   <==  [% XX>=0, XY>0 %];
IR REM1;
  XX REM XY >= 0   <==  [% XX>=0, XY>0 %];
IR REM2;
  XX REM XY =<0   <==  [% XX=<0 %];

1->ISASSDC("GCD");
1->ISCDMM("GCD");
WHEN [% 0<XX, XX=<XY %],  GCD(XX,XY-XX) ==> GCD(XX,XY);
WHEN [% XY>0, XX REM XY=0 %],  GCD(XX,XY) ==> XY;

IR GCD1
  GCD(XU,XX) = GCD(XV,XX)
  <==  [% XX/=0, (XU-XV) REM XX = 0 %];

CHNG(CHNG(XA,XA\XI,XJ),XA\XJ,XI)  ==>  XCHNG(XA,XI,XJ);

WHEN [% XU<XI,XV<XI %],
     XCHNG(XA,XU,XV)<<XI,XJ>>  ==>  XA<<XI,XJ>>;
WHEN [% XU<XI,XV>XJ %],
     XCHNG(XA,XU,XV)<<XI,XJ>>  ==>  XA<<XI,XJ>>;
WHEN [% XU>XJ,XV<XI %],
     XCHNG(XA,XU,XV)<<XI,XJ>>  ==>  XA<<XI,XJ>>;
WHEN [% XU>XJ,XV>XJ %],
     XCHNG(XA,XU,XV)<<XI,XJ>>  ==>  XA<<XI,XJ>>;

IR CHNG1;
  XF(CHNG(XA,XX,XI)<<XI,XJ>>, XZ)
  <==  [% XF(XX,XZ), XF(XA<<XI+1,XJ>>, XZ) %];
TDRULES;
IR CHNG2;
  XF(CHNG(XA,XX,XJ)<<XI,XJ>>,XZ)
  <==  [% XF(XA<<XI,XJ-1>>,XZ), XF(XX,XZ) %];
TDRULES;

IR XCHNG1
  XF(XCHNG(XA,XU,XV)<<XI,XJ>>, XZ)
  <==  [% XU=XJ, XV>XJ, XF(XA<<XI,XJ-1>>,XZ), XF(XA\XV,XZ) %];
TDRULES;
IR XCHNG2
  XF(XCHNG(XA,XU,XV)<<XI,XJ>>, XZ)
  <==  [% XV=XJ, XU>XJ, XF(XA<<XI,XJ-1>>,XZ), XF(XA\XU,XZ) %];
TDRULES;
```

```
IR XCHNG3
  XF(XCHNG(XA,XU,XV)<<XI,XJ>>, XZ)
  <==  [% XU=XI, XV<XI, XF(XA<<XI+1,XJ>>,XZ), XF(XA\XV,XZ) %];
TORULES;
IR XCHNG4
  XF(XCHNG(XA,XU,XV)<<XI,XJ>>, XZ)
  <==  [% XV=XI, XU<XI, XF(XA<<XI+1,XJ>>,XZ), XF(XA\XU,XZ) %];
TORULES;
IR XCHNG5
  XF(XCHNG(XA,XU,XV)<<XI,XJ>>, XZ)
  <== [% XI=<XU,XU=<XJ,XI=<XV,XV=<XJ, XF(XA<<XI,XJ>>,XZ) %];
TORULES;
```

```
1->ISCOMM("EQSEQ");
1->ISTRANS("EQSEQ");
IR EQSEQ10
  EQSEQ(XX,XX) <== NIL;
IR EQSEQ12
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>)
  <==  [% XI=<XJ, XJ-XI=XM-XK, XA\XJ=XB\XM,
          EQSEQ(XA<<XI,XJ-1>>,XB<<XK,XM-1>>) %];
IR EQSEQ13
  XA\XJ=XB\XM
  <== [% (EX [XI XK])(XI=<XJ & XM-XK=XJ-XI &
                        EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>)) %];
IR EQSEQ14
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% XI=<XJ+1, XM-XK=XJ-XI,
         EQSEQ(XA<<XI,XJ+1>>,XB<<XK,XM+1>>) %];

IR EQSEQ15
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XL>>)
  <== [% XJ-XI = XL-XK,
         (FA XU)(0=<XU&XU=<XJ-XI =>> XA\(XI+XU)=XB\(XK+XU)) %];
IR EQSEQ16
  XA\XU=XB\XV
  <== [% (EX [XI XJ XK XL])(XI=<XU & XU=<XJ & XK=<XV & XV=<XL &
              XJ-XI = XL-XK & XU-XI = XV-XK &
              EQSEQ(XA<<XI,XJ>>,XB<<XK,XL>>)) %];

IR ISIN10
  ISIN(XA<<XI,XJ>>,XB<<XK,XL>>)
  <== [% (EX XM)(XK=<XM & XM+XJ-XI=<XL &
                  EQSEQ(XA<<XI,XJ>>,XB<<XM,XM+XJ-XI>>)) %];
IR ISIN11
  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% XJ-XI=<XM-XK, EQSEQ(XA<<XI,XJ>>,XB<<XM-XJ+XI,XM>>) %];
IR ISIN12
  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)
  <== [% ISIN(XA<<XI,XJ>>,XB<<XK,XM-1>>) %];
IR ISIN13
  EQSEQ(XA<<XI,XJ>>,XB<<XK,XL>>)
  <== [% (EX XM)(XM=<XK & XJ-XI=<XL-XM &
                 ISIN(XA<<XI,XJ>>,XB<<XM,XL>>)) %];
```

```
IR ISIN14
  ISIN(XA<<XI,XJ>>,XB<<XK,XM>>)
  <==  [% XJ-XI=<XM-XK+1, ISIN(XA<<XI,XJ>>,XB<<XK,XM+1>>),
          NOT(EQSEQ(XA<<XI,XJ>>,XB<<XM+1-XJ+XI,XM+1>>))  %];

IR ISIN20
  NOT(ISIN(XA<<XI,XJ>>,XB<<XK,XL>>))  <==  [% XJ-XI>XL-XK %];

IR ORD12
  ORDERED(XA<<XI,XJ>>)
  <== [% XI<XJ, XA\(XJ-1)=<XA\XJ, ORDERED(XA<<XI,XJ-1>>) %];
IR ORD13
  XA\(XJ-1) =< XA\XJ
  <== [% (EX XI)(XI<XJ & ORDERED(XA<<XI,XJ>>)) %];
IR ORD14
  ORDERED(XA<<XI,XJ>>)
  <== [% XI<XJ+1, ORDERED(XA<<XI,XJ+1>>) %];

1->ISCOMM("PERM");
1->ISTRANS("PERM");
IR PERM10
  PERM(XX,XX) <== NIL;
IR PERM11
  PERM(XA<<XI,XJ>>,XB<<XI,XJ>>)
  <==  [% XA\XI=XB\XI, PERM(XA<<XI+1,XJ>>,XB<<XI+1,XJ>>) %];
IR PERM12
  PERM(XA<<XI,XJ>>,XB<<XI,XJ>>)
  <==  [% XA\XJ=XB\XJ, PERM(XA<<XI,XJ-1>>,XB<<XI,XJ-1>>) %];
IR PERM13
  PERM(XCHNG(XA,XU,XV)<<XI,XJ>>, XA<<XI,XJ>>)
  <==  [% XI=<XU,XU=<XJ, XI=<XV,XV=<XJ %];

IR MEMB5
  MEMB(XU,XV<>XW) <== [% MEMB(XU,XV) %];
IR MEMB6
  MEMB(XU,XV<>XW) <== [% MEMB(XU,XW) %];
```

Appendix 4: Programs verified

This appendix contains a list of the programs verified by our system. The list is not complete but it is indicative of the upper level of the verifier's performance. We have verified some programs using both inclusion statements and assertions, and in such cases we show the program with each type of specification for comparison.

The examples in King's thesis have become benchmarks; our system can verify all of these examples but we have only shown the more interesting of them here.

Most of these programs required at least some help from the user in the verification process.

Counting-up factorial

This is a simple "counting-up" program to compute the factorial function. It is basically the same program which was used as an example in the introduction. An alternative way to describe the program using virtual programs would be to use the function prod(j,k) = j*(j+1)*...*k, defined recursively, instead of the function div.

```
FUNCTION FAC3 N=>R;
VARS I;
 1->I; 1->R;
$LOOP:
  LOOPIF I=<N THEN R*I->R; I+1->I; CLOSE;
$FINISH:
END;


VARS N0;
[N0]->INITARGS;

ISTAT BODY FAC3;
 VIRT FACTORIAL(N)->R; N+1->I;
UNOER N>=0;
  WRT [R I];

ISTAT LOOP TO [FINISH];
 VIRT R*FACTORIAL(N) DIV FACTORIAL(I-1) -> R; N+1->I;
UNOER 0<I &  I=<N+1;
  WRT [R I];
```

Recursive and iterative factorial

This is the program discussed in Sections 2.5.2 and 3.3.2. We show the program twice: once described by virtual programs alone, and once using inductive assertions as well.

```
FUNCTION FAC4V N=>R;
 1->R;
$ LOOP:
  LOOPIF N>0 THEN N-1->N; R+N*FAC4V(N)->R; CLOSE;
$ FINISH:
END;


VARS N0;
[N0]->INITARGS;

ISTAT BODY REC FAC4V
 VIRT FACTORIAL(N)->R; 0->N;
UNDER N>=0;
  WRT [N R];

ISTAT LOOP TO [FINISH]
 VIRT R+FACTORIAL(N)-1->R; 0->N;
UNDER N>=0;
  WRT [N R];
```

```
FUNCTION FAC4A N0 => R;
VARS N;
    N0->N; 1->R;
$ LODP:
    ASSERT R+FACTORIAL(N)-1 = FACTORIAL(N0)  &  N>=0 ;
    LOOPIF N>0 THEN N-1->N; R+N*FAC4A(N)->R CLOSE;
$ FIN:
ENO;
```

```
VARS NI;
[NI]->INITARGS;

ISTAT BOOY REC FAC4A
 VIRT FACTORIAL(N0)->R; 0->N
UNOER N0>=0
  WRT [N R];
```

Greatest common divisor

This is a program to find the greatest common divisor of two positive integers without using division. It is only one of several greatest common divisor programs the system has verified, chosen to illustrate the difference between the two ways of describing it. We first show the inductive assertions for the program, then inclusion statements which describe each loop separately, and finally simpler inclusion statements which all terminate at the end of the function body and do not preserve the loop structure.

```
FUNCTION GCD3A M N => R;
$START:
   ASSERT M=M0 & N=N0 & M0>0 & N0>0;
$LOOP:
   LOOPIF M/=N THEN
   $L1:
      ASSERT GCD(M,N)=GCD(M0,N0) & M>0 & N>0;
      LOOPIF M>N THEN M-N->M CLOSE;
   $L2:
      ASSERT GCD(M,N)=GCD(M0,N0) & M>0 & N>0;
      LOOPIF N>M THEN N-M->N CLOSE;
   CLOSE;
   M->R;
$OUT:
   ASSERT R=GCD(M0,N0);
END;


VARS M0 N0;
[M0 N0]->INITARGS;
```

Greatest common divisor (continued)

```
FUNCTION GCD3V M N => R;
$LOOP:
 LOOPIF M/=N THEN
   $INLP1:
    LOOPIF M>N THEN M-N->M CLOSE;
   $INLP2:
    LOOPIF N>M THEN N-M->N CLOSE;
 CLOSE;
$OUT:
 M->R;
END;


VARS M0 N0;
[M0 N0]->INITARGS;

ISTAT BODY GCD3V;
VIRT  GCD(M,N)->R;
UNDER M>0 & N>0;
WRT   [R];

ISTAT LOOP TO [OUT];
VIRT  GCD(M,N)->M;
UNDER M>0 & N>0;
WRT   [M];

ISTAT INLP1 TO [INLP2];
VIRT  (ANY "M1")(0<M1 & M1=<M & M1=<N & (M-M1)REM N=0) -> M;

UNDER M>0 & N>0;
WRT   [M N];

ISTAT INLP2 TO [LOOP];
VIRT  (ANY "N1")(0<N1 & N1=<N & N1=<M & (N-N1)REM M=0) -> N;

UNDER M>0 & N>0;
WRT   [M N];
```

Greatest common divisor (continued)

```
FUNCTION GCD3V2 M N => R;
$LOOP:
 LOOPIF M/=N THEN
   $INLP1:
    LOOPIF M>N THEN M-N->M CLOSE;
   $INLP2:
    LOOPIF N>M THEN N-M->N CLOSE;
 CLOSE;
 M->R;
$OUT:
END;


VARS M0 N0;
[M0 N0]->INITARGS;

ISTAT BODY GCD3V2
VIRT  GCD(M,N) -> R
UNDER M>0 & N>0
WRT   [R];

ISTAT LOOP TO [OUT]
VIRT  GCD(M,N) -> R
UNDER M>0 & N>0
WRT   [R];

ISTAT INLP1 TO [OUT]
VIRT  GCD(M,N) -> R
UNDER M>0 & N>0
WRT   [R];

ISTAT INLP2 TO [OUT]
VIRT  GCD(M,N) -> R
UNDER M>0 & N>0
WRT   [R];
```

91-function

This function was discussed in Section 2.5.1 and its proof was shown in Section 6.1.

```
FUNCTION FN91 N => R;
  IF N>100 THEN N-10 ELSE FN91(FN91(N+11)) CLOSE -> R;
END;

VARS N0;
[N0]->INITARGS;

ISTAT BODY REC FN91;
VIRT  IF N>100 THEN N-10 ELSE 91 CLOSE -> R;
UNDER TRUE;
WRT   [R]
```

Ackermann's function

This example shows how Ackermann's function can be computed directly for m=<3. If m>3 the virtual program is undefined but this is irrelevant as this path is never taken in the proof.

```
FUNCTION ACK M N => R;
IF M=0 THEN N+1
ELSE IF N=0 THEN ACK(M-1,1)
     ELSE ACK(M-1,ACK(M,N-1)) CLOSE CLOSE -> R;
END;


VARS M0 N0 INFINITY;
[M0 N0]->INITARGS;

ISTAT BODY REC ACK;
 VIRT IF M=0 THEN N+1 ELSE
      IF M=1 THEN N+2 ELSE
      IF M=2 THEN 2*N+3 ELSE
      IF M=3 THEN 2↑(N+3)-3
         ELSE INFINITY CLOSE CLOSE CLOSE CLOSE -> R;
UNDER 0=<M & M=<3 &  0=<N;
  WRT [R];
```

Tree-searching function

This function searches the tips of a binary tree for a 1 as described in Section 2.5.4. If it succeeds it terminates by using a JUMPOUT (escape).

```
FUNCTION TREEJ T => TV;
VARS WON;
 JUMPOUT(LAMBOA; $FND: ENO, 0) -> WON;
  FUNCTION SEARCH1 T;
  $START:
   IF ATOM(T) THEN
      IF T=1 THEN TRUE->TV; WON() CLOSE
   ELSE SEARCH1(FRONT(T)); SEARCH1(BACK(T))
   CLOSE;
  $OUT:
  ENO;

 FALSE->TV;
 SEARCH1(T);
ENO;


VARS TO;
[TO]->INITARGS;

ISTAT START TO [OUT REC FNO]
VIRT  IF MEMB(1,FRINGE(T)) THEN TRUE->TV; WON()
         ELSE GOTO OUT CLOSE
UNOER TRUE
WRT   [TV];

ISTAT BOOY TREEJ
VIRT  MEMB(1,FRINGE(T))->TV
UNOER TRUE
WRT   [TV];
```

Matcher

This program determines whether the array B occurs as a subarray of the array A. The proof produced by the system is shown in Section 6.2. We show here the program specifications using virtual programs and inductive assertions successively.

```
FUNCTION MATCHV A M B N => BISINA;
VARS I J ;
 FALSE->BISINA;
 0->I;
$LOOPI:
 LOOPIF I=< M-N THEN
    1->J;
  $LOOPJ:
    LOOPIF J=<N THEN
       IF A\(I+J)/=B\(J) THEN GOTO BREAKJ CLOSE;
       J+1->J;
    CLOSE;
  $ENDLOOPJ:
    TRUE->BISINA; GOTO BREAKI;
  $BREAKJ:
    I+1->I;
 CLOSE;
$BREAKI:
END;


OECARRAY A [1 M];
OECARRAY B [1 N];
[%A,"M",B,"N"%]->INITARGS;

ISTAT BODY MATCHV;
VIRT  IF ISIN(B<<1,N>>,A<<1,M>>)
      THEN TRUE ELSE FALSE CLOSE -> BISINA;
UNDER 0=<N &  N=<M;
WRT   [BISINA];

ISTAT LOOPI TO [BREAKI]
VIRT  IF ISIN(B<<1,N>>,A<<I+1,M>>)
      THEN TRUE->BISINA;
      ELSE M-N+1->I CLOSE;
      UNDER 0=<I &  I=<M-N+1 &  0=<N;
WRT   [BISINA];

ISTAT LOOPJ TO [ENDLOOPJ BREAKJ];
VIRT  IF EQSEQ(B<<J,N>>,A<<I+J,I+N>>)
      THEN N+1->J; GOTO ENDLOOPJ
      ELSE GOTO BREAKJ CLOSE;
UNDER  1=<J &  J=<N+1;
WRT   [I];
```

Matcher (continued)

```
FUNCTION MATCHA A M B N => BISINA;
VARS I J;
ASSERT 0=<N & N=<M;
   FALSE->BISINA;
   0->I;
$LOOP:
   LOOPIF I=<M-N THEN
      1->J;
    $LOOPJ:
      ASSERT NOT(ISIN(B<<1,N>>,A<<1,I+N-1>>))
          &  EQSEQ(B<<1,J-1>>,A<<I+1,I+J-1>>)
          &  NOT(BISINA)
          & 1=<J & J=<N+1 & 0=<I & I=<M-N & 0=<N & N=<M;

      LOOPIF J=<N THEN
         IF A\(I+J)/=B\J THEN GOTO BREAKJ CLOSE;
         J+1->J
      CLOSE;
      TRUE->BISINA;
      GOTO OUT;
    $BREAKJ:
      I+1->I
   CLOSE;
$OUT:
   ASSERT BISINA<=>ISIN(B<<1,N>>,A<<1,M>>);
END;


VARS M0 N0;
DECARRAY A0 [1 M0];
DECARRAY B0 [1 N0];
[% A0, "M0", B0, "N0" %] -> INITARGS;
```

King's example 6

This program moves the largest element of an array to the top by successive interchanges and was discussed in Sections 2.5.5 and 3.3.1. Actually, we have been unable to use the system to completely verify the program with the specifications given as virtual programs because of a difficulty in reasoning about existential quantifiers. However, we were able to complete the proof using inductive assertions.

```
FUNCTION K6V A N;
VARS I;
    2->I;
$LOOP:
    LOOPIF I=<N THEN
       IF A\(I-1)>A\I THEN A\I,A\(I-1)->A\I->A\(I-1) CLOSE;
       I+1->I;
    CLOSE;
END;


VARS N0;
DECARRAY A0 [1 N0];
[%A0,"N0"%]->INITARGS;

ISTAT LOOP TO [K6VEND];
VIRT  (ANYARR "A1")(EQSEQ(A1<<1,I-2>>,A<<1,I-2>>) &
                    PERM(A1<<I-1,N>>,A<<I-1,N>>)  &
                    A1<<I-1,N-1>> =< A1\N) -> A;
UNDER 2=<I &  I=<N+1;
WRT   [A N];

ISTAT BODY K6V;
VIRT  (ANYARR "A1") (PERM(A1<<1,N>>,A<<1,N>>)  &
                     A1<<1,N-1>> =< A1\N ) -> A;
UNDER N>=1;
WRT   [A N];
```

King's example 6 (continued)

```
FUNCTION K6A A N;
VARS I;
   ASSERT A=A0 & 1=<N;
   2->I;
$LOOP:
   ASSERT  PERM(A<<1,I-1>>, A0<<1,I-1>>)
        &  A<<1,I-2>> =< A\(I-1)
        &  EQSEQ(A<<I,N>>,A0<<I,N>>)
        &  2=<I & I=<N+1;
   LOOPIF I=<N THEN
      IF A\(I-1)>A\I THEN A\I, A\(I-1) ->A\I ->A\(I-1) CLOSE;
      I+1->I
   CLOSE;
$OUT:
   ASSERT PERM(A<<1,N>>,A0<<1,N>>)
        & A<<1,N-1>> =< A\N ;
END;


VARS N0;
DECARRAY A0 [1 N0];
[% A0, "N0" %] -> INITARGS;
```

King's example 7: Exchange sort

The system verified this program automatically except for one manual application of INTERM. The specifications for this program (and the next one) are given as inductive assertions only, but we have shown how these could be translated into equivalent inclusion statements.

```
FUNCTION K7 A N;
VARS I NOTORO;
   ASSERT A=A0 & 1=<N;
   TRUE->NOTORO;
$L1:
   LOOPIF NOTORO THEN
      2->I; FALSE->NOTORO;
    $L2:
      ASSERT NOTORO OR ORDEREO(A<<1,I-1>> )
           & PERM(A<<1,N>>,A0<<1,N>>)
           & 2=<I & I=<N+1;

      LOOPIF I=<N THEN
         IF A\(I-1) > A\I THEN
            A\I, A\(I-1) -> A\I -> A\(I-1);
            TRUE->NOTORD
         CLOSE;
         I+1->I
      CLOSE
   CLOSE;
$OUT:
   ASSERT ORDEREO(A<<1,N>>)
       &  PERM(A<<1,N>>,A0<<1,N>>);
END;


VARS N0;
DECARRAY A0 [1 N0];
[% A0, "N0" %] ->INITARGS;
```

King's example 9: Insertion sort

The proof of this program is given in Section 6.3.

```
FUNCTION K9 A N;
VARS I J K X;
   ASSERT A=A0 & 2=<N;
   1->I;
$L1:
   LOOPIF I<N THEN
      A\I->X; I->K; I+1->J;
   $L2:
      ASSERT 1=<I & I=<K & K<J & J=<N+1
           & I<N
           & X=A\K
           & (I=1 OR A\(I-1) =< A<<I,N>> )
           & A<<I,J-1>> >= X
           & ORDERED(A<<1,I-1>>)
           & PERM(A<<1,N>>,A0<<1,N>>);

      LOOPIF J=<N THEN
         IF X>A\J THEN A\J->X; J->K CLOSE;
         J+1->J;
      CLOSE:
      A\I->A\K; X->A\I; I+1->I;
   CLOSE;
$OUT:
   ASSERT ORDERED(A<<1,N>>) & PERM(A<<1,N>>,A0<<1,N>>);
END;


VARS N0;
DECARRAY A0 [1 N0];
[% A0,"N0" %]->INITARGS;
```

Invert a permutation "in place"

This program, presented by Knuth (1972, p172), was verified after the remainder of the thesis had been completed. The proof took about two weeks to find and check, and because of its length was done in several sessions. The proof is loosely based on that given by Burstall (1974) and uses both loop invariants and virtual programs in the program's specifications. It depends on the fact that a permutation can be decomposed into disjoint cycles (as does the program). The proof is complete except for the fact that the inverse of a permutation is also a permutation and that the relation of two elements being in the same cycle (INCYCLE) is an equivalence relation.

Doing this proof substantiated our belief that it is preferable to use definitions involving explicit quantifiers rather than recursive definitions, particularly since relations such as INCYCLE1 are quite difficult to define recursively, and our initial attempts to use such a definition in the proof failed.

In the following pages we show the program with its specifications and the rules used in the proof. Several of the predicates and functions used actually require A0 and N as additional arguments, but since these remain constant throughout they have been omitted.

Permutation (continued)

```
FUNCTION INVERT A N;
VARS I J K M;
$START:
   ASSERT  N>=1 & A=A0 & ISPERM(A0<<1,N>>);
   N->M;
$LOOPM:
   ASSERT  0=<M & M=<N & ISPERM(A0<<1,N>>)  &  (FA "Q")(
      (M<Q & Q=<N  =>>  A\Q = INV(A0)\Q )  &
      (1=<Q & Q=<M & INVERTED(Q,M)  =>>  A\Q = 0-INV(A0)\Q )  &
      (1=<Q & Q=<M & NOT(INVERTED(Q,M))  =>>  A\Q = A0\Q ));

   LOOPIF M>=1 THEN
      A\M->I;
      IF I<0 THEN (0-I)->A\M
      ELSE  M->K:
        $LOOPI:
            LOOPIF I/=M THEN
               A\I->J; (0-K)->A\I; I->K; J->I;
            CLOSE;
            K->A\M;
         $FIN:
      CLOSE;
      M-1->M:
   CLOSE;
$OUT:
   ASSERT  (FA "Q")(1=<Q & Q=<N  =>>  A\Q = INV(A0)\Q);
END;


VARS N0;
DECARRAY A0 [1 N0];
[% A0,"N0" %] -> INITARGS;


ISTAT LOOPI TO [FIN]
VIRT  (ANYARR "B")((FA "R")((R=M =>> B\R = INV(A0)\R ) &
            (R/=M & INCYCLE1(I,R,M) =>> B\R = 0-INV(A0)\R )  &
            (NOT(INCYCLE1(I,R,M)) =>> B\R = A\R) ))  -> A;
UNDER 1=<K & K=<N & 1=<M & M=<N & I=A0\K  & ISPERM(A0<<1,N>>) &
      (EX "V")(0=<V & V<LEN(I) & M=ITFN(I,V)) &
      (FA "S")(INCYCLE1(I,S,M) =>> A\S=A0\S)
WRT   [A M];
```

Permutation (continued)

```
INCYCLE1(XK,XQ,XM)
  ==>  (EX [XU XV])(0=<XU & XU=<XV & XV<LEN(XK) &
                    XQ=ITFN(XK,XU) & XM=ITFN(XK,XV));

INCYCLE(XP,XQ)
  ==>  (EX XU)(0=<XU & XU<LEN(XP) & XQ=ITFN(XP,XU));

INVERTED(XQ,XM)  ==>  (EX XI)(XM<XI & XI=<N & INCYCLE(XI,XQ));

IR 14
  ISPERM(XA<<XM,XN>>)
  <==  [% (FA XQ)(XM=<XQ & XQ=<XN =>> (XM=<XA\XQ & XA\XQ=<XN)),
          (FA [XP XQ])(XM=<XP & XP=<XN & XM=<XQ & XQ=<XN &
                       XP/=XQ  =>>  XA\XP /= XA\XQ ) %];
IR 15
  XM =< XA\XQ
  <==  [% (EX XN)(XM=<XQ & XQ=<XN & ISPERM(XA<<XM,XN>>)) %];
IR 16
  XA\XQ =< XN
  <==  [% (EX XM)(XM=<XQ & XQ=<XN & ISPERM(XA<<XM,XN>>)) %];
IR 17
  XA\XP /= XA\XQ
  <==  [% (EX [XM XN])(XM=<XP & XP=<XN & XM=<XQ & XQ=<XN &
                       XP/=XQ & ISPERM(XA<<XM,XN>>) ) %];

IR 20
  INV(XA)\XI = XJ  <==  [% XA\XJ = XI %];
IR 21
  ISPERM(INV(XA)<<XM,XN>>)  <==  [% ISPERM(XA<<XM,XN>>) %];

ITFN(XP,0)  ==>  XP;
WHEN [% XX/=0 %],  ITFN(XP,XX*LEN(XP)+XY)  ==>  ITFN(XP,XY);
ITFN(ITFN(XP,XJ),XI)  ==>  ITFN(XP,XI+XJ);
A0\XP  ==>  ITFN(XP,1);

IR L0
  1 =< LEN(XM)  <==  NIL;
IR L1
  XI = LEN(XP)  <==  [% ITFN(XP,XI)=XP, 0<XI, XI=<LEN(XP) %];
IR L2
  XI = XJ
  <==  [% (EX XP)(ITFN(XP,XI)=ITFN(XP,XJ) &
           0=<XI & XI<LEN(XP) & 0=<XJ & XJ<LEN(XP)) %];
IR L3
  XI = LEN(XA\XP)  <==  [% XI = LEN(XP) %];
```

Permutation (continued)

```
IR 101
  INCYCLE(XM,XM)  <==  NIL;
IR 102
  INCYCLE(XQ,XM)  <==  [% INCYCLE(XM,XQ) %];
1->ISTRANS("INCYCLE");

IR 105
  1=<XQ
  <==  [% (EX [XK XM])(1=<XK & XK=<N & 1=<XM & XM=<N &
                       INCYCLE1(XK,XQ,XM) ) %];
IR 106
  XQ=<N
  <==  [% (EX [XK XM])(1=<XK & XK=<N & 1=<XM & XM=<N &
                       INCYCLE1(XK,XQ,XM)) %];
```

Appendix 5: Listing of matcher

```
[LIB ASSOC].LIBRARY.COMPILE;

VARS INST INSTL APSUBST1 SUBXS ASUBXS ASUBXS1 ACSUBXS ACSUBXS1;


COMMENT'*****************************************************
*   INSTANCE IS THE ASSOCIATIVE, COMMUTATIVE MATCHER -
*   CALLED BY
*                INSTANCE(TERM,PATTERN) => SUBSTITUTION-LIST
*****************************************************@;


FUNCTION INSTANCE TERM PAT;
 INST(TERM,PAT,ASSNIL());
END;



FUNCTION INST TERM PAT SIG;
VARS PATN PATL TERML S1 GENSUBXS;
 IF ISVAR(PAT) THEN  COMMENT 'VARIABLE@;
   ASSOC(PAT,SIG)->S1;
   IF S1=UNDEF THEN [%UPDASSOC(TERM,PAT,SIG)%]
   ELSEIF EQX(S1,TERM) THEN [%SIG%]
   ELSE NIL CLOSE
 ELSEIF ISPRIM(PAT) THEN  COMMENT 'CONSTANT@;
   IF EQX(PAT,TERM) THEN [%SIG%] ELSE NIL CLOSE;
 ELSE  COMMENT 'FUNAP@;
   FUNNAME(PAT)->PATN; FUNARGS(PAT)->PATL;
   IF ISASSOC(PATN) THEN
      IF ISCOMM(PATN) THEN ACSUBXS ELSE ASUBXS CLOSE -> GENSUBXS;
      IF ISFUNAP(TERM) THEN
         IF PATN=FUNNAME(TERM) THEN
            INSTL(FUNARGS(TERM),PATL,PATN,SIG,GENSUBXS)
         ELSE INSTL([%TERM%],PATL,PATN,SIG,GENSUBXS)
         CLOSE
      ELSE INSTL([%TERM%],PATL,PATN,SIG,GENSUBXS)
      CLOSE
   ELSEIF ISFUNAP(TERM) AND PATN=FUNNAME(TERM) THEN
      FUNARGS(TERM)->TERML;
      INSTL(TERML,PATL,PATN,SIG,SUBXS);
      IF ISCOMM(PATN) AND NOT(EQX(HD(TERML),HD(TL(TERML))))
                      AND NOT(EQX(HD(PATL), HD(TL(PATL)))) THEN
         <> INSTL(REV(TERML),PATL,PATN,SIG,SUBXS);
      CLOSE
   ELSE NIL CLOSE
 CLOSE
END;
```

```
FUNCTION INSTL TERML PATL OP SIG GENSUBXS => SIGS;
COMMENT'*****************************************************
*   MATCH THE ARG-LIST TERML AGAINST PATL IN ALL POSSIBLE WAYS,
*   DEPENDING ON THE PROPERTIES OF OP AS DESCRIBED BY GENSUBXS
*****************************************************@;
VARS PAIR PAIRS ISIG ISIGS IDENT;
 IF PATL.NULL THEN
   IF TERML.NULL THEN [%SIG%] ELSE NIL CLOSE->SIGS;
 EXIT;
 IDENTOF(PATN)->IDENT; NIL->SIGS;
 LOOPIF TERML/=NIL AND EQX(HD(TERML),IDENT)
     THEN TL(TERML)->TERML CLOSE;
  GENSUBXS(TERML,OP)->PAIRS;
  LOOPIF PAIRS/=NIL THEN DEST(PAIRS)->PAIRS->PAIR;
    INST(FRONT(PAIR),HD(PATL),SIG)->ISIGS;
    LOOPIF ISIGS/=NIL THEN DEST(ISIGS)->ISIGS->ISIG;
      INSTL(BACK(PAIR),TL(PATL),OP,ISIG,GENSUBXS) <> SIGS -> SIGS;
    CLOSE;
  CLOSE;
ENO;


FUNCTION SUBXS XL OP;
 IF NULL(XL) THEN NIL ELSE XL::NIL CLOSE;
ENO;


FUNCTION ASUBXS XL OP => XS;
COMMENT'*****************************************************
*   XS IS THE SET OF ALL POSSIBLE (INITIAL) PARTITIONS
*   OF THE ARGLIST XL INTO A TERM AND REMAINING ARGLIST,
*   EG,  ASUBXS([A B],".") = [ (1,[A B]), (A,[B]), (A.B,NIL) ],
*   WHERE   (IDENT(".")=1)
*****************************************************@;
VARS N LB;
 LENGTH(XL)->N;
 IF IDENT=UNDEF THEN 1 ELSE 0 CLOSE -> LB;
 [% LOOPIF N>=LB THEN ASUBXS1(XL,N); N-1->N CLOSE %] -> XS;
ENO;


FUNCTION ASUBXS1 XL N;
 CONSPAIR(
    ABBREV([%LOOPIF N THEN XL.DEST->XL; N-1->N CLOSE%],OP,IDENT),
    XL);
ENO;


FUNCTION ACSUBXS XL OP => XS;
COMMENT'*****************************************************
*   LIKE ASUBXS, BUT FINDS THE SET OF ALL POSSIBLE SUBBAGS,
*   EG,  ACSUBXS([A B],".") = [ A,[B]), (B,[A]) ],
*   WHERE  IDENTOF(".")=UNDEF .
*****************************************************@;
 NIL->XS;
 ACSUBXS1(REV(XL),NIL,NIL);
ENO;
```

```
FUNCTION ACSUBXS1 XL S1 S2;
COMMENT 'SINCE XL IS SORTED, EQUAL ELEMENTS ARE ADJACENT@;
VARS X L;
 IF NULL(XL) THEN
    UNLESS IDENT=UNDEF AND NULL(S1)
    THEN CONSPAIR(ABBREV(S1,OP,IDENT),S2)::XS->XS CLOSE;
 EXIT;
 DEST(XL)->XL->X; X::S1->L;
 LOOPIF XL/=NIL AND EQX(X,HD(XL)) THEN X::L->L; TL(XL)->XL CLOSE;
 LOOPIF L/=S1 THEN ACSUBXS1(XL,L,S2); DEST(L)->L->X; X::S2->S2
 CLOSE;
 ACSUBXS1(XL,L,S2);
END;


FUNCTION ABBREV XL OP ID;
COMMENT'*****************************************************
*   CONSTRUCT THE TERM WITH FUNCTION OP,  NORMALIZED ARGLIST XL,
*   AND IDENTITY ID.  OP MUST BE ASSOCIATIVE.
*   ABBREV IS ONLY REALLY REQUIRED WHEN A NORMALIZING FUNCTION
*   WHICH KNOWS ABOUT INVERSES IS USED.
*************************************************************@;
 IF NULL(XL) THEN ID
 ELSEIF NULL(TL(XL)) THEN HD(XL)
 ELSE MKFUNAP(OP,XL) CLOSE;
END;


FUNCTION UPDASSOC CPT SUB ASS;
COMMENT 'A CONSTRUCTIVE UPDATER FOR ASSOC,
         ASS MUST BE UNDEFINED AT SUB!;
  ASSCONS(ASSFAIL(ASS),ASSEQ(ASS),ASSLCONS(CPT,SUB,ASSOF(ASS)));
END;
```

SCHOOL OF ARTIFICIAL INTELLIGENCE
UNIVERSITY OF EDINBURGH

Memorandum: MIP-R-104
Date: July, 1974

Subject:- The correctness of the Schorr-Waite list marking algorithm

Author:- R. Topor

1. Introduction

We present here a proof of the correctness of an algorithm, due to Schorr and Waite (1967) and to Deutsch (Knuth 1968, p.417), which traces and marks an arbitrary list structure, and which can be used for the first stage of garbage collection. The algorithm is of interest because of the clever way it avoids using a stack. by manipulating pointers within the structure, restoring them all at the end. The general problem of data structure updating is a difficult one, and work on it has been done by Burstall (1972), Morris (1972), Poupon and Wegbreit (1972) and Kowaltowski (1973), though we have not used any of their methods.

The correctness proof of the Schorr-Waite algorithm given here is simpler than those given by Poupon and Wegbreit, or Kowaltowski. Our proof is factored into properties of the algorithm itself, and properties of the data structure upon which it operates. In fact, one can use these latter properties to prove correct two simpler versions of the algorithm, one using recursion and the other an explicit stack. The proof involves not inductive assertions (Floyd 1967), but mathematical induction on the size of the structure to be marked. It can be formalized using the method of Burstall (1974), itself a variant of Manna (1969).

The method of proof was suggested by a hint in Knuth (1968, p.420).*

John Reynolds (1974) has used similar techniques in his treatment of Tarjan's algorithm (1972).

I am grateful to Rod Burstall and Gordon Plotkin for several helpful suggestions. The work was supported by a Commonwealth Scholarship.

2. The problem of marking

We are given a set of nodes, i.e. list cells, each node containing two fields (hd,tl) which may contain atoms or pointers to other nodes, a mark bit (m), and, for the Schorr-Waite algorithm, a flag bit (f); we are also given a particular starting node z_0. The structure may contain shared and cyclic sublists. Initially all the mark bits are set to 0. The problem is to set the mark bit of every node reachable by a finite sequence of hd's and tl's from z_0 to 1.

The main difficulty in proving that an algorithm to solve this problem is correct is that the natural technique of structural induction does not work; because lists may be cyclic there is no sense in which hd(x) is less than x (they may even be the same node). An added difficulty in verifying the Schorr-Waite algorithm is showing that, apart from the mark bits, the structure is the same at the end as it was at the start, despite the destructive updating of the nodes. To overcome these difficulties we introduce the following concepts before discussing the actual algorithms.

3. Properties of marked list structures

Let C be the set of all nodes, A the set of all atoms, and $\sum: C \to (C \cup A)^2 \times \{0,1\}^2$ the set of possible machine states.

Definition We define functions hd_σ, $tl_\sigma: C \to C \cup A$, $m_\sigma, f_\sigma: C \to \{0,1\}$ by $hd_\sigma c = x_1$ where $\sigma(c) = \langle x_1, x_2, b_1, b_2 \rangle$,

$tl_\sigma c = x_2$ " "

$m_\sigma c = b_1$ " "

$f_\sigma c = b_2$ " "

We further define Marked: $\sum \to 2^C$ by $Marked_\sigma = \{c \in C: m_\sigma c = 1\}$.

The subscript σ is often omitted from these functions where this can be done unambiguously.

The/

The key idea is to define the set of unmarked nodes reachable from a given node. This is the purpose of the following definitions.

Definition The predicate ispath: $\sum \times(C \cup A)^2 \to \{true,false\}$, is defined by $ispath_\sigma(x,y)$ iff there exists a finite sequence $x_0,x_1,\ldots,x_n$, $n\geq 0$ of distinct nodes in C such that $x_0=x, x_n=y$, for $0\leq i\leq n$, $m_\sigma x_i=0$, and for $0\leq i<n$, $x_{i+1}=hd_\sigma x_i$ or $x_{i+1}=tl_\sigma x_i$. Such a sequence is called a path (w.r.t. σ).

We can now define nodes: $\sum \times(C \cup A)\to 2^C$ by
$nodes_\sigma(x)=\{y \in C \mid ispath_\sigma(x,y)\}$.
We assume that C, and hence $nodes_\sigma(x)$, is a finite set throughout.

Fact 1 For all x, σ, $nodes_\sigma(x)=\emptyset$ if $x \in (A \cup Marked_\sigma)$,
$=\{x\} \cup nodes_\sigma(hd_\sigma x) \cup nodes_\sigma(tl_\sigma x)$ otherwise.

Proof Clearly if $x \in A \cup Marked$, $ispath(x,y)=false$ for all $y \in C$, so $nodes(x)=\emptyset$.
Suppose $x \notin A \cup Marked$, and let $y \in nodes(x)$.
If $y=x$ then $y \in \{x\} \cup nodes(hd\ x) \cup nodes(tl\ x)$.
Otherwise ispath(x,y), i.e. $x=x_0,x_1,x_2,\ldots,x_n=y$, $n\geq 1$, where $x_1=hd\ x$ or $x_1=tl\ x$, is a path. Suppose $x_1=hd\ x$, then ispath(hd x,y), so $y \in nodes(hd\ x)$.
Now let $y \in \{x\} \cup nodes(hd\ x) \cup nodes(tl\ x)$.
If $y=x$, since $x \notin A \cup Marked$, ispath(x,x), so $y \in nodes(x)$.
Otherwise $y \in nodes(hd\ x)$, say, and $hd\ x=x_0,x_1,\ldots,x_n=y$, $n\geq 0$ is a path.
If for some i, $0<i\leq n$, $x_i=x$, then ispath(x,y) and $y \in nodes(x)$.
Otherwise $x,hd\ x=x_0,x_1,\ldots,x_n=y_n$ is a path, and again, $y \in nodes(x)$.

Fact 2 For all x,y, σ, if $y \in nodes_\sigma(x)$, then $m_\sigma x=0$.

Proof Immediate from the definitions of nodes and ispath.

Definition For $\sigma 1$, $\sigma 2 \in \sum$, define $\sigma 1 \sqsubseteq \sigma 2$ if $Marked_{\sigma 2} \subseteq Marked_{\sigma 1}$, $hd_{\sigma 2}=hd_{\sigma 2}$, $tl_{\sigma 1}=tl_{\sigma 2}$ and $f_{\sigma 1}=f_{\sigma 2}$, i.e. if there are fewer unmarked nodes in $\sigma 1$ than in $\sigma 2$. (In cases like this we say $Marked_{\sigma 2} \subseteq Marked_{\sigma 1}$ and/

and $\sigma 1 = \sigma 2$ otherwise.) The relation $\sqsubseteq$ is clearly a partial order.

Fact 3 For all x, $\sigma 1$, $\sigma 2$, if $\sigma 1 \sqsubseteq \sigma 2$, then $nodes_{\sigma 1}(x) \subseteq nodes_{\sigma 2}(x)$.

Proof Let $y \in nodes_{\sigma 1}(x)$, i.e. $ispath_{\sigma 1}(x,y)$.
But as $hd_{\sigma 1}=hd_{\sigma 2}$, $tl_{\sigma 1}=tl_{\sigma 2}$ and $m_{\sigma 1}x_i=0 \rightarrow m_{\sigma 2}x_i=0$, $ispath_{\sigma 2}(x,y)$.
Hence $y \in nodes_{\sigma 2}(x)$.

Definition We define mark: $C \times \Sigma \rightarrow \Sigma$ by $mark_x(\sigma) = \sigma'$ where $\sigma' = \sigma$ (i.e. $hd_\sigma = hd_{\sigma'}$, $tl_\sigma = tl_{\sigma'}$, and $f_\sigma = f_{\sigma'}$) except that $m_{\sigma'}y=1$ iff $m_\sigma y=1$ or $y \in nodes_\sigma(x)$. This concept is important for describing the effects of the marking algorithms.

Fact 4 For all x,y,σ, $nodes_{mark_x(\sigma)}(y) \cup nodes_\sigma(x) = nodes_\sigma(y) \cup nodes_\sigma(x)$, i.e. marking one node's descendants and then the other's is the same as marking them all together.

Proof a) $mark_x(\sigma) \sqsubseteq \sigma'$, so $nodes_{mark_x(\sigma)}(y) \subseteq nodes_\sigma(y)$ (Fact 3).
Thus $nodes_{mark_x(\sigma)}(y) \cup nodes_\sigma(x) \subseteq nodes_\sigma(y) \cup nodes_\sigma(x)$.

b) let $z \in nodes_\sigma(y)$, and let $y=y_0,y_1,\ldots,y_n=z$, $n \geq 0$ be a path.
If for all i, $0 \leq i \leq n$, $m_{mark_x(\sigma)}(y_i)=0$, $ispath_{mark_x(\sigma)}(y,z)$, so
$z \in nodes_{mark_x(\sigma)}(y)$.
Otherwise, let $j = \mu_{i \geq 0}\, m_{mark_x(\sigma)}(y_i)=1$.
Since $m_\sigma y_j=0$, $y_i \in nodes_\sigma(x)$, and $ispath_\sigma(x,y_j)$.
Clearly $ispath_\sigma(y_j,z)$, so $ispath_\sigma(x,z)$ and $z \in nodes_\sigma(x)$.
Hence $nodes_\sigma(y) \subseteq nodes_{mark_x(\sigma)}(y) \cup nodes_\sigma(x)$, and the result follows.

Fact 5 For all x, $\sigma 1$, $\sigma 2$, if $Marked_{\sigma 2} = Marked_{\sigma 1} \cup \{x\}$, $x \notin Marked_{\sigma 1}$, and $\sigma 1 = \sigma 2$ otherwise, then $nodes_{\sigma 1}(x) = \{x\} \cup nodes_{\sigma 2}(hd\ x) \cup nodes_{\sigma 2}(tl\ x)$.

Proof a) since $\sigma 2 \sqsubseteq \sigma 1$, $nodes_{\sigma 2}(hd\ x) \subseteq nodes_{\sigma 1}(hd\ x)$ and
$nodes_{\sigma 2}(tl\ x) \subseteq nodes_{\sigma 1}(tl\ x)$.
So $\{x\} \cup nodes_{\sigma 2}(hd\ x) \cup nodes_{\sigma 2}(tl\ x) \subseteq \{x\} \cup nodes_{\sigma 1}(hd\ x) \cup nodes_{\sigma 1}(tl\ x)$
$= nodes_{\sigma 1}(x)$ ($x \notin A \cup Marked_{\sigma 1}$, Fact 1).

b) let $y \in nodes_{\sigma 1}(x)$, i.e. $x=x_0,x_1,\ldots,x_n=y$, $n \geq 0$ is a path (w.r.t. $\sigma 1$).
If/

If n=0, y=x and y $\in$ {x} $\cup$ nodes$_{\sigma 2}$(hd x) $\cup$ nodes$_{\sigma 2}$(tl x).
Otherwise, x_1=hd x, say, so ispath$_{\sigma 2}$(hd x,y) and y $\in$ nodes$_{\sigma 2}$(hd x).
Thus nodes$_{\sigma 1}$(x) $\subseteq$ {x} $\cup$ nodes$_{\sigma 2}$(hd x) $\cup$ nodes$_{\sigma 2}$(tl x).

4. Two simple algorithms

The facts we have derived so far are sufficient and necessary to prove the correctness of the following two ancestors of the Schorr-Waite algorithm. The first, recursive program is perhaps the simplest possible marking algorithm. The second is obtained by replacing the recursion by an explicit stack. The proofs of these programs are analogous to the one we are about to give, only simpler as no destructive assignment is involved. We omit these proofs here.

Program 1

```
Mk(Z) <= if not(atom(Z)) and m(Z)=0
            then m(Z)<-1; Mk(hd(Z)); Mk(tl(Z)) fi;
Mk(Z0);
```

Program 2

```
Start: Z<-Z0; S<-empty;
Loop : while not(atom(Z)) and m(Z)=0
          do(m(Z)<-1; S<-push(Z,S); Z<-hd(Z));
       if S≠empty then Z<-pop(S); Z<-tl(Z); goto Loop fi;
Finish:
```

5. The Schorr-Waite algorithm

This algorithm saves the stack of Program 2 in the already marked nodes. The f-bit is used to determine whether the back pointer to the next node of the stack is in the hd or the tl of the current node (X). Initially all the f-bits are 0. To handle this destructive updating we introduce the following:

Definition/

Definition For $\sigma 1, \sigma 2 \in \Sigma$, define $\sigma 1 \sim \sigma 2$ if $\text{Marked}_{\sigma 1} = \text{Marked}_{\sigma 2}$ and for all $x \notin \text{Marked}_{\sigma 1}$, $\sigma 1(x) = \sigma 2(x)$, i.e. $\sigma 1$ and $\sigma 2$ only differ on marked nodes.

Fact 6 If $\sigma 1 \sim \sigma 2$ then for all x, $\text{nodes}_{\sigma 1}(x) = \text{nodes}_{\sigma 2}(x)$.

Proof Clearly $\text{ispath}_{\sigma 1}(x,y)$ iff $\text{ispath}_{\sigma 2}(x,y)$ and the result follows.

We now give the algorithm. The comma as a connective for the multiple assignment statement means that all left and right hand values are calculated, and then the assignments are carried out simultaneously.

```
Start: Z<-Z0; X<-nil;
P1:    while not(atom(Z)) and m(Z)=0
          do(m(Z)<-1, hd(Z)<-X, Z<-hd(Z), X<-Z);
P2:    if X=nil then goto Finish fi;
       if f(X)=0
          then f(X)<-1, hd(X)<-Z, tl(X)<-hd(X). Z<-tl(X); goto P1
          else f(X)<-0, tl(X)<-Z, X<-tl(X), Z<-X; goto P2 fi;
Finish:
```

Before stating and proving the correctness criterion we still need to introduce a little more notation. We write $f[x \mapsto y]$ for the function g defined by $g(z) \Leftarrow$ if $z=x$ then y else $f(z)$. Following Burstall (1974) we write "P: $X_1=x_1$, $X_2=x_2$,..." as an abbreviation for "there exists a stage of the execution when the computation is at label P, and the identifiers X_i have the values x_i". When we say "by computation" during a proof we mean "by observing the effects of the assignments on the state vector". We introduce an imaginary variable, Store, whose value is the current state.

Theorem If Start: $Z_0=z_0$, Store=σ where for all $y \in \text{nodes}_{\sigma}(z_0)$ $f_{\sigma}y=0$, then Finish: $Z_0=z_0$, Store=$\text{mark}_{z_0}(\sigma)$, i.e. all the nodes reachable from z_0 are marked but otherwise unchanged.

Proof/

Proof Letting $z=z_0$ and x=nil, this theorem can be seen to follow immediately from the following lemma.

Lemma If P1: Z=z, X=x, Store=σ where for all $y \in nodes_\sigma(z)$ $f_\sigma y=0$, then P2: Z=z, X=x, Store=$mark_z(\sigma)$.

Proof The proof is by course of values induction on the size of $nodes_\sigma(z)$. Notice how the induction hypothesis is used twice, corresponding to the two recursive calls of Program 1.

Basis $|nodes_\sigma(z)|=0$, i.e. $z \in A \cup Marked_\sigma$, the first test fails, and the result is immediate.

Step $|nodes_\sigma(z)| \neq 0$, i.e. $z \notin A \cup Marked_\sigma$.
Let $hd_\sigma z=u$, $tl_\sigma z=v$ and $\sigma[z \mapsto \langle hd_\sigma z, tl_\sigma z, 1, f_\sigma z\rangle]=\sigma'$, i.e. $\sigma'=\sigma$ except that $m_{\sigma'}z=1$.

Since $z \notin A \cup Marked_\sigma$, the body of the while statement is executed once, yielding P1: Z=u, X=z, Store=$\sigma 1=\sigma[z \mapsto \langle x, tl_\sigma z, 1, f_\sigma z\rangle]$.
Now, $nodes_{\sigma 1}(u)=nodes_{\sigma 1}(hd_\sigma z)$
$=nodes_{\sigma'}(hd_\sigma z)$ ($\sigma 1 \sim \sigma'$, Fact 6)
$\subseteq nodes_\sigma(hd_\sigma z)$ ($\sigma' \sqsubseteq \sigma$, Fact 3)
$\subseteq nodes_\sigma(z)$ (Fact 1).
But $m_{\sigma 1}z=1$, so $z \notin nodes_{\sigma 1}(u)$ (Fact 2).
Since $z \in nodes_\sigma(z)$, $nodes_{\sigma 1}(u) \subsetneq nodes_\sigma(z)$, and $|nodes_{\sigma 1}(u)| < |nodes_\sigma(z)|$.

We can now use the induction hypothesis, i.e. the lemma with z=u, x=z, $\sigma=\sigma 1$, to obtain

P2: Z=u, X=z, Store=$\sigma 2=mark_u(\sigma 1)$.

Now, as $z \neq nil$, and $f_{\sigma 2}z=0$, by computation we have

P1: Z=v, X=z, Store=$\sigma 3=\sigma 2[z \mapsto \langle u, hd_{\sigma 2}z, m_{\sigma 2}z, 1\rangle]$.
Thus $hd_{\sigma 3}z=hd_\sigma z$, $tl_{\sigma 3}z=hd_{\sigma 2}z=x$, and $\sigma 3 \sim \sigma'$.
As before, $nodes_{\sigma 3}(v)=nodes_{\sigma 3}(tl_\sigma z)$
$=nodes_{\sigma'}(tl_\sigma z)$ (Fact 6)
$\subseteq nodes_\sigma(tl_\sigma z)$ ($\sigma' \sqsubseteq \sigma$, Fact 3)
$\subseteq nodes_\sigma(z)$ (Fact 1),

and/

and as $m_{\sigma 3}z=1$, $z \notin nodes_{\sigma 3}(v)$, so $|nodes_{\sigma 3}(v)| < |nodes_{\sigma}(z)|$.

Again we use the induction hypothesis to obtain

P2: Z=v, X=z, Store=$\sigma 4=mark_v(\sigma 3)$.

This time z≠nil, but $f_{\sigma 4}z=1$, so computation yields

P2: Z=z, X=x, Store $\sigma 5=\sigma 4[z \mapsto \langle hd_{\sigma 4}z, v, m_{\sigma 4}z, 0\rangle]$.

Thus $hd_{\sigma 5}z=u=hd_{\sigma}z$, $tl_{\sigma 5}z=v=tl_{\sigma}z$ and $f_{\sigma 5}z=0$. Since applying $mark_x$ to σ does not affect hd_{σ}, tl_{σ} or f_{σ}, $hd_{\sigma 5}=hd_{\sigma}$, $tl_{\sigma 5}=tl_{\sigma}$ and $f_{\sigma 5}=f_{\sigma}$.

It only remains to show that $\sigma 5=mark_z(\sigma)$, i.e. that $Marked_{\sigma 5}=nodes_{\sigma}(z)$.

$Marked_{\sigma 5}=\{z\} \cup nodes_{\sigma 1}(u) \cup nodes_{\sigma 3}(v)$

$=\{z\} \cup nodes_{\sigma 1}(u) \cup nodes_{mark_u(\sigma 1)}(v)$ ($\sigma 3 \sim \sigma 2=mark_u(\sigma 1)$, Fact 6)

$=\{z\} \cup nodes_{\sigma 1}(u) \cup nodes_{\sigma 1}(v)$ (Fact 4)

$=\{z\} \cup nodes_{\sigma'}(u) \cup nodes_{\sigma'}(v)$ ($\sigma 1 \sim \sigma'$, Fact 6)

$=nodes_{\sigma}(z)$ (Fact 5)

This completes the proof.

An alternative method of proof is to first prove Program 2, a purely constructive program using this method. and then, using the techniques of Milner (1971) or Hoare (1972) to show that the Schorr-Waite algorithm simulates Program 2. In particular the representation function, Rep, for the stack in the Schorr-Waite algorithm is defined by

```
Rep(X) <= if X=nil then empty
          elseif f(X)=1 then Rep(tl(X))
          else push(X,Rep(hd(X))).
```

However the resulting proof by this approach is longer than the one we have given.

6./

6. References

Burstall, R.M. (1972) Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7 (eds. B. Meltzer and D. Michie), Edinburgh: University Press, pp. 23-50.

Burstall, R.M. (1974) Program proving as hand simulation with a little induction. To appear in Proceedings of IFIP Congress '74, Stockholm, Sweden, August 5.10.

Floyd, R.W. (1967) Assigning meanings to programs. Proceedings of Symposium in Applied Mathematics, AMS Vol. 19, 19-32.

Knuth, D.E. (1968) Fundamental Algorithms. Addison-Wesley.

Hoare, C.A.R. (1972) Proof of correctness of data representations. Acta Informatica 1, 271-281.

Kowaltowski, T. (1973) Correctness of programs manipulating data structures. Ph.D. Thesis. Berkeley.

Manna, Z. (1969) The correctness of programs. J.C.S.S., 3, 2, 119-127.

Milner, R. (1971) An algebraic definition of simulation between programs. CS 205, Stanford University.

Morris, J.H. Jr. (1972) Verification-oriented language design. Tech. Rep. 7, Dept. of Computer Science, Berkeley.

Poupon, J. and Wegbreit, B. (1972) Covering function. Centre for Research in Computing Technology, Harvard University.

Reynolds, J. (1974) Unpublished work.

Schorr, H. and Waite, W. (1967) An efficient machine-independent procedure for garbage collection in various list structures. C.A.C.M. 10, 8.

Tarjan, R. (1972) Depth-first search and linear graphs. SIAM J. of Comp. 1.