Stanford Artificial Intelligence Laboratory
Memo AIM-278

March 1976

Computer Science Department
Report No. STAN-CS-76-549

Automatic Program Verification V:
VERIFICATION-ORIENTED PROOF RULES for ARRAYS, RECORDS AND POINTERS

by

David Luckham *and* **Norihisa Suzuki**

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-278

March 1976

Computer Science Department
Report No. STAN-CS-76-549

Automatic Program Verification V:

VERIFICATION-ORIENTED PROOF RULES for ARRAYS, RECORDS AND POINTERS

by

David Luckham and Norihisa Suzuki

ABSTRACT

A- practical method is presented for automating in a uniform way the verification of Pascal programs that operate on the standard Pascal data structures ARRAY, RECORD, and POINTER. New assertion language primitives are introduced for describing computational effects of operations on these, data structures. Axioms defining the semantics of the new primitives are given. Proof rules for standard Pascal operations on pointer variables are then defined in terms of the extended assertion language. Similar rules for records and arrays are special cases. An extensible axiomatic rule for the Pascal memory allocation operation, NEW, is also given.

These rules have. been implemented in the Stanford Pascal program verifier. Examples illustrating the verification of programs which operate on list structures implemented with pointers and records are discussed. These include programs with side-effects.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 . The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, 07 the V. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 2 2 1 5 1 .

1. INTRODUCTION

This paper presents axiomatic proof rules for standard PASCAL operations on the data structures ARRAY, RECORD and POINTER, Axiomatic semantics for these data structures have been given in some form in previous publications ([Hoare & Wirth], [Burstall], [Spitzen & Wegbreit]). However, here, our emphasis is on the notion of a proof rule. That is, we are interested in defining proof rules for operations on these structures that are suitable for addition to the existing set of proof rules employed by current automatic verifiers -- this we call verification oriented semantics. These rules not only define the semantics of operations on the data structures axiomatically. They are also programmable reduction rules suitable for automating a significant part of the search for proofs of programs that operate on complex data structures.

The main problem from the point of view of extending the present verifiers, is to be able to cope with certain forms of the assignment statement. The semantic definition of assignment given in [Hoare 69] is entirely adequate for assignment to a variable of any arbitrary type. In this paper we are concerned with finding verification rules for assignment in the case when the left hand side is an expression containing operations which select a substructure of a data structure. For example, array assignment rules given in [King], [Igarashi, London, & Luckham] (henceforth called [ILL]), and [Suzuki a] define the semantics of A[I]←E. Here the index I "selects" or picks out an element of the array data structure A, so the meaning is different from assignment to the variable A itself-- a specified part of the value of A is changed!

We shall give rules for standard Pascal operations such as X↑.F←Y where X is a pointer to a record with field F. Rules for these kinds of operations are needed in

order, to improve, program verification methods to a point where certain classes of complex programs such as garbage collectors and schedulers can be verified.

The idea presented here is to generalize the rule in standard use for assignment to an array element. This leads to a single scheme which defines proof rules for assignment to substructures of array, record and pointer structures as special cases. In addition, the allocation operation, NEW(X), whereby new structures can be created during a computation, needs to be given a verification oriented rule. We do this here at the same time.

Section 2 presents an overview of both the way proof rules can be used in automating verification, and of how considerations similar to those which led to the array rule will lead to. our generalization of it for records and pointers. We feel that it is reasonable to say something about the use of the proof rules since some of our **decisions** are based on facilitating implementation. However we do rely on earlier papers [ILL, Suzuki b] for full details about verification systems. Section 3 gives the general definitions of the extended assertion language and the most general form of the new proof rules. Section 4 is devoted to illustrating how a verifier with these rules can be used to obtain proofs of properties of programs which operate on tree structures built up from pointers and records. It is shown here that our extended verification system is capable of proving such properties as " program A does not introduce loops into list structure L" for actual programs containing about a page of Pascal code.

In this paper we omit formal justification of our rules. Normally, this would take the 'form of a soundness proof. A model of PASCAL computations would be defined and then it would be shown that the proof rules describe state **transformations** of, the model. Instead we rely on the motivation in Section 2 to

convince the reader that our formal rules do correspond to his intuitive understanding of the PASCAL semantics.

2. MOTIVATION

The reasoning which leads us to our proof rules can be paraphrased as follows. First we have to "know" intuitively what the PASCAL operations do; that is, what transformations they make to data structures. We extend the standard assertion language (i.e.. Pascal Boolean expressions with the addition of quantifiers and defined relations --see [ILL,' Suzuki b]) so that it contains expressions which formally represent data structures and transformations of data structures. These new assertion language expressions are called data structure representations. Then we can give formal proof rules for Pascal operations in terms of such representations. The representations themselves have semantic definition rules which permit simplifications to be made automatically, This enables proofs of simple programs to be completely automated. Below we outline this' reasoning by giving first the "intuitive" transformation rule for an operation on a structure, then the new expressions that we add to the assertion language to represent the transformation and the semantics of the expressions, and then the formal proof rule for that operation. We deal in succession with the cases of Arrays, Records, and finally, Pointers. This should clarify the general definitions of representations and proof rules in Section 3. We begin here with a short discussion of verification oriented rules in general.

2.1 Reduction Rules.

Axiomatic semantic rules within Hoare's weak logic of programs [Hoare 69,71, ILL] are nearly all of the form

$$\frac{A \ , \ B}{C}$$

meaning "if A and B are both true (the premisses of the rule) then C is also true (conclusion)". Here, A, B, C, are either Boolean formulas or statements about programs. The latter kind of statement has the form P{S}Q where P and Q are Boolean formulas and S is a program part (i.e. a sequence of Pascal statements). P and Q are the input and output specifications for S. In the deduction rule, C is always a statement about a program part.

We can regard a deduction as taking place by applying a rule "downwards". However, such a rule is employed "upwards" as a problem reduction rule in a typical verifier [ILL]. This means that if some problem C' matches C in the sense that C' = C$\propto$ where $\propto$ is a substitution of actual parameters for formal parameters, then A$\propto$ and B$\propto$ will be generated as "reduced" problems to be solved. This reduction process can be continued until all the reduced problems are purely logical formulas and do not contain any program statements. These formulas are called Verification Conditions (VC's). The reader is referred to [ILL] for examples of problem reduction and generation of VC's.

2.2 Forwards Rules and Backwards Rules.

The semantic meaning of the assignment statement is defined by axioms in Hdare's system. For example, assignment to a simple variable may be defined by (AVF stands for Assignment to a Variable Forwards):

AVF., $P(X) \wedge X = X0 \{X \leftarrow E\} P(X0) \wedge X = E \Big|_{X0}^{X}$

where $E \Big|_{X0}^{X}$ denotes the substitution of X0 for X in E.

The axiom AVF is a true statement of the Logic of Programs for ail formulas P. Intuitively, this axiom describes the way $X \leftarrow E$ changes the state of any computation:

It says, suppose $P \wedge X = X0$ is true of the state before $X \leftarrow E$. Then after executing $X \leftarrow E$, two things will be true: (a) the value of X will change to $E \Big|_{X0}^{X}$ and (b) true statements about the value of X before assignment are still true of the old value X0 after.

We call this axiom a "forwards" rule because the postcondition (after execution) shows how the precondition (before execution) is changed. Such rules are not the easiest to implement in automatic verification systems because of the equality terms $X = E \Big|_{X0}^{X}$ in the post condition, The basic problem is the question of when-to substitute $E \Big|_{X0}^{X}$ for X in any formulas that may get generated later on in

the process. It is easier to avoid the generation of equalities altogether. So, in verification systems we often use "backwards" axioms like AVB (from **[Hoare]**).

AVB. **P(E) {X←E}P(X)**

where P(E) is P with E substituted for all occurrences of X. This is a "backwards" rule: it states that if P(X) is to be true after **X←E** is executed then P(E) must be true before. This is equivalent to saying that the effect of **X←E** will be to give X the value E. The forwards and backwards versions of the rules are equivalent, and the verification conditions produced by verifiers using either version are also equivalent.

A verifier, given a problem ENTRY **{S1;...;Sn}EXIT,** and using backwards axioms **will** work backwards in the following sense. Starting with EXIT it will deduce **(using** either upwards or backwards rules) what has to be true before statement Sn, and from that it will deduce what must be true before Sn-1, and so on.

In the following we shall develop backwards rules since they are easier to implement.

2.3 Assignment to Array Elements.

Now consider an axiomatic semantic rule for assignment to an element of an array (Assignment to Array Backwards) given in terms of an informal assertion language:

AAB. If **I=J** then **P(E)** else **P(A[J]){A[I]←E}P(A[J])**

We might ail agree (given that we understand the meaning of "if-then-else") that this defines the meaning of "A[I]←E". The rule states what must be true of the computation state of a program before performing A[I]←E if P(A[J]) is to be true after. The semantics is defined by the change in the computation state. Rule AAB is a scheme in that it holds for all formulas .P. However, if we add this rule to a verifier, we have the complication that if we are trying to verify, say ENTRY {B;A[I]←E}P(A[J]), an application AAB will leave us to verify

(1). ENTRY(B) (if I=J then P(E) else P(A[J])).

And we will not know at the time (1) is generated whether I=J or not. The information required to determine if I=J is most likely contained in the preceding program B.

Thus rule AAB requires the assertion language to contain array and index variables, and conditionals. in addition, the reduction rules will have to allow for conditional assertions.

Nested conditional assertions grow exponentially, and it is advisable for implementation to replace them by an explicit representation in the assertion language of the the change to A resulting from A[I]←E. To achieve this, we have introduced assertion language expressions that represent the result of selector and assignment operations on arrays. It should be emphasized that the expressions represent structures resulting from operations.

Syntax of REWRITE and SELECTOR expressions for Arrays:

REWRITE: <A, [I], E>

SELECTOR: [J]

where A is an array of elements of type T, I and J are indices,

and E is an expression of type T.

Intuitively, the rewrite expression represents the array obtained from A by assigning E to A[I]. And <A,[I],E>[J] represents the Jth element of this array. The two kinds of expressions can be concatenated together (see example 1 below), and the rewrites may be nested to represent the result of sequences of operations on A.

These assertion language expressions obey the following rules which define their semantics:

SEM1. <A, [I], E>[J] = E if I=J,

<A, [I], E>[J] = A[J] if I≠J,

The verficiation-oriented rule for assignment to arrays may now be given using the extended assertion language:

V1. P(<A,[I],E>){A[I]←E}P(A)

where ail occurences of A in P(A) are replaced by <A,[I],E> to form P(<A,[I],E>).

Note the special case of V1: P(<A,[I],E>[J]){A[I]←E}P(A[J]).

This is our version of AAB.

Let us see how the rules **V1 and SEM1** work on a simple example.

```
EXAMPLE 1.    1. A[K]←I
              2. A[A[K]]←E
            EXIT P(A[I]).
```

We want the exit assertion to be true after the two operations. Successive applications of **(V1)** state that **P(<A,[A[K]],E>[I])** must be true before instruction 2, **and P(<<A,[K],I>,[<A,K,I>[K]],E>[I])** must be true before 1. Using **SEM1** this last assertion reduces to P(E).

Essentially, the introduction of the REWRITE expressions into the assertion language, is to represent the changes in the data **stucture** that occur as the result of assignment to an array element. The semantics of programming language statements assigning to array elements are then defined in terms of such changes by rule **V1.** The rule **SEM1** enables us to simplify expressions containing rewrites and selectors when the values of indices are determined. It is clear that both rules are easy to implement so that both the construction of the representations and their simplification can be automated.

The notation for REWRITE used here is due to **[Hoare** and Wirth]; different notation appears in [King]. One of the nice features of this notation is its compact nesting property for representing successive assignments.

2.4 Assignment to Record Fields.

An assignment, **R.F←E** where R is a record with a field F, changes a record data

structure in exactly the same way as assignment to an array element changes an array. Analogous assertions and rules are used to define the semantics of assignment to a record field. We describe them briefly here.

Syntax of REWRITE and SELECTOR expressions for Records:

REWRITE: **<R, .F, E>**

SELECTOR: **.F**

where R is a record, F is an identifier of a field of **R** of type T, and E is an expression of **type** T.

The semantics of these new assertion language expressions are given by:

SEM2. **<R, .F, E>.G** = E if **F=G,**

<R, .F, E>.G = R.G if **F≠G.**

The verification proof rule for assignment to record fields is:

V2. P(<R, .F, E>){R.F←E}P(R)

2.5 Assignment to Dereferenced Pointers.

Let us now define similar axiomatic rules for assignment to dereferenced pointers,-i.e. assignments of the form **X↑←E.** Intuitively, **X↑←E** means that the value in

the memory location to which X points is changed to E.

We might try to define the semantics of such statements by a backwards rule such as

APB. if X=Y then P(E) else P(Y↑){X↑←E}P(Y↑)

The rule is an obvious backwards way saying that if X and Y point to the same memory location (i.e. X=Y) before X↑←E, then Y↑=E afterwards.

This rule resembles the intuitive backwards array rule, AAB, with X playing the role of an index I. In AAB, I picks out an element of the array A. However, in this case we do not have a name in the assertion language for the set of values X can point to (i.e., reference). So the first thing we shall do is to introduce names for such sets of values called REFERENCE CLASSES (the early Pascal definition contains the concept of a reference class [Wirth]). Of course, a reference class is unbounded, but it can be accessed and parts of it selected in exactly the same way as an array. So the notation we shall use for representing computations on reference classes will be very similar (in fact the differences are merely to distinguish them from operations on arrays). For example, if P*REF is a reference class then P*REF⊂X⊃ will denote the value that X points to (i.e. the same thing as Xt). The result of X↑←E can be represented by <P*REF,⊂X⊃,E>. In this notation the round brackets are analogous to the square brackets for indexing arrays.

Thus we extend the assertion language in order to represent computations involving assignment to dereferenced pointers as follows.

For each pointer type declaration,

TYPE name1 = ↑name2

we add P*name2 to the assertion language. This is the name of the finite reference class of elements of'type name2 that exist at the start of a computation.

Syntax of REWRITE and SELECTOR expressions for Reference Classes.

REWRITE: <C, ⊂X⊃, E>

SELECTOR: ⊂X⊃

where C is a reference class of elements of type T, X is a pointer of type ↑T, and E is an expression of type T.

These expressions satisfy semantic rules similar to previous ones:

SEM3. <C, ⊂X⊃, E>⊂Y⊃ = E if X=Y

<C, ⊂X⊃, E>⊂Y⊃ = C⊂Y⊃ if X≠Y

The verification rule for assignment to dereferenced pointers is:

v3 a. P(<P*name2, ⊂X⊃, E>){ X↑←E}P(P*name2)

and

b. P(<P*name2, ⊂X↑⊃, E>⊂Y⊃){X↑←E}P(Y↑)
for ail occurrences in P of Y of type namel.

The reader may note that our extension of the 'assertion language has

introduced different notation for the same thing; Y↑ and P*name2⊂Y⊃ both represent the value Y points to. If the verifier uniformly eliminates one notation in favour of the other, we shall need only one of the V3 rules.

Let us see how this rule will work on a typical "side-effects" example.

EXAMPLE 2.

```
TYPE A = ↑B;
VAR X,Y:A;
1. Y ←X;
2. X↑←1;
3. Y↑←2;
EXIT X↑=2.
```

This example has a side effect in the sense that instruction 3 mentions only the value Y↑ but also changes the value X↑.

If the exit is true after 3, then by (V3)b. <P*B,⊂Y⊃,2>⊂X⊃ = 2 must be true before 3. By (V3)a, <<P*B,⊂X⊃,1>,⊂Y⊃,2>⊂X⊃=2 must hold before 2. But now the simple assignment rule for variables, P(X){ Y←X}P(Y), tells us that <<P*B,⊂X⊃,1>,⊂X⊃,2>⊂X⊃=2 has to hold on entry. This is easily seen to reduce to 2=2 by SEM3.

2.6 Storage Allocation.

A reference class is indefinitely extendible by the Pascal allocation operation, NEW(X). The intuitive meaning of NEW(X) is that a memory ceil which has not previously occurred in the computation is appended to the reference class P*name2, and the value of X is changed so that X "points to" this new cell. The value of X↑ is undefined? It is assumed that such a new 'ceil always exists. This semantics is defined

by means of memory mapping functions in [Hoare & Wirth].

Our assertions must be able to represent such extensions, so we introduce the notation P*name2U{X'} to represent the reference class of X extended by the operation NEW(X), where X' is a "new" identifier. More generally, DU(X'} represents an extension of the class represented by D. We refer to "U" as the extension operation on data structures. We now have to see if this addition to the assertion language is sufficient to permit the definition of a proof rule for allocation.

The problem facing us here is to define a semantic proof rule which states how an arbitrary assertion about a computation state is affected by allocation. Our rule must express both of the effects of NEW(X), namely the extension of the reference class and the "newness" of X. Let us discuss these two aspects seperateiy.

First, suppose a reference class has a representation of the form, <P*T, ⊂Y⊃, E>. After NEW(X) its representation will be <P*T, ⊂Y⊃, E>U{X'} where X' is an identifier not occurring in any expression so far (i.e. a new identifier). But the newness of X' clearly implies that <P*TU{X'}, ⊂Y⊃, E> also represents the same structure. More generally, we have:

SEM4 If <D,S,E> represents a reference class and X' is a new identifier, then <D,S,E>U{X'} and <DU{X'},S,E> represent the same reference class.

So a first approximation, to a backwards rule for allocation, expressing only the extension of a reference class (analogous to the backwards rule for assignment) is:

$$Q(P*T \cup \{X'\})\{NEW(X)\}Q(P*T)$$

where X' is a new identifier, and P*T is the name of the reference class of elements of type-of Xt, and X does not occur in Q.

Secondly, how does an allocation NEW(X) affect an assertion about X, say Q(X)? The intended semantics is that X is given a "new" value X' which is distinct from any previous pointer, and nothing else in the state is changed. Any arbitrary new value X' may be allocated to X. Ignoring the extension of P*T, these properties are expressed by the following backwards rule:

$$\wedge(Yi \in SET_OF\ P*T)(X' \neq Yi) \supset Q(X')\{NEW(X)\}Q(X)$$

where X' is a new identifier, and SET-OF P*T is the set of ail pointer expressions of type-of X that do not contain X'.

This rule states that if Q(X) is to be true after NEW(X), then Q(X') must be true of any "new" X' before.

We may combine the two rules above as follows.

NEW B. $$\wedge(Yi \in SET_OF\ P*T)(X' \neq Yi) \supset Q\left|{}^{P\#T}_{P\#T \cup \{X'\}}\right|^{X}_{X'}\{NEW(X)\}Q$$

where P*T is the name of the reference class of elements of type-of XT, X' is a new identifier, and SET-OF P*T is the set of ail pointer expressions of type-of X that do not contain X'.

This rule assumes the axioms SEM4. In addition we have further axiomatic properties of the extension operation:

SEM5. DU(Y)⊂P▪D⊂X⊃ if X≠Y, and is undefined if X=Y,

where D is a representation of a reference class.

We cannot implement NEWB as it stands because SET-OF P▪T is too large. The verification rule for NEW in Section 3 is weaker but can be strengthened by additional axioms from the user.

2.7, Sequences of selectors.

So far we have dealt with assignments in which the left side contains only one selector operation. Pascal allows sequences of selector operations. We have to extend the assertion language still further by introducing sequences of selectors in order to represent the data structure changes made by such assignments.

For example, consider X↑.F↑.G. This is a selector sequence that would be applicable to a list of records where the 'F field of each record was a pointer to the next record in the list. We can compute the representation as follows. P▪N⊂X⊃ represents XT; P▪N⊂X⊃.F represents X↑.F which is another pointer; so P▪N⊂P▪N⊂X⊃.F⊃ represents X↑.F↑ and the representaton of the entire sequence above is P▪N⊂P▪N⊂X⊃.F⊃.G. This is a sequence of the form P▪N⊂Z⊃.G where Z is not a simple pointer variable, but is a representation of a data structure of type pointer. So our selectors will not be as simple as before.

Simultaneously, the set of rewrite expressions that will now be used to represent data structures within the assertion language must also be extended. Thus,

the change to the reference class P*N that occurs when X↑.F↑.G←E is executed can be represented by the rewrite, <P*N, ⊂P*N⊂X⊃.F⊃.G, E>. As we see from this example, the syntax of rewrites must be extended to permit representations of the form <X,S,E> where S is a selector sequence.

It should be noted that the rule for assignment with a single selector on the left is not sufficient to express the general assignment even if we introduce dummy program variables, For example, we could try to rewrite X↑.F↑.G ←E as Y←X↑.F↑;Y.G←E. However,. in the second case, E is placed in the G field of a new copy of X↑.F↑, whereas in the first case E is placed directly into the original record.

3. PROOF RULES FOR OPERATIONS ON DATA STRUCTURES.

In this section we define proof rules for assignment statements with expressions involving data structure selectors in the most general case. The rule for assignment presented here can be regarded as defining the semantics of assignment. In the case of dereferenced pointers it fills in a gap in the axiomatic semantics of Pascal assignment in [Hoare & Wirth]. We shall also present a rule for storage allocation which is not complete in any reasonable sense, but which represents a compromise between a logically complete rule and what is computationally feasible for automating proofs. It can be extended by the user to handle any particular problem.

First, we must define the extensions of the standard assertion language (c.f.[ILL] section 2) that have been introduced expressly for the purpose of making statements about complex data structures (i.e. structures containing identifiable substructures).

3.1 New Assertion Language Primitives

Notation: We will use $\otimes$ to denote concatenation.

ϕ denotes the empty sequence.

Complex data structures are represented by Assertion Language expressions of the form $\langle A,I,E \rangle$ and $A \otimes J$ where A and E are themselves data structure representations, and I and J are sequences of applicable selectors. Intuitively, $\langle A,I,E \rangle$ represents "the structure obtained from A by replacing the substructure of A'

selected by I, with E".A@J represents "the substructure of A selected by J". This notation generalizes the notation. for arrays used by earlier writers ([McCarthy], [King], [Hoare & Wirth]). We will first define the syntax of the representations.

Terminology: A TYPE-NAME is any identifier introduced as the name of a type by a Pascal type declaration.

DEFINITION (reference class identifier)

For each pointer type declaration, TYPE T=↑T0; where TO is a type 'identifier, we introduce a reference class identifier P*T0 for the reference class of T0.

Intuitively, P*T0 represents an unbounded set of data structures of type T0 that pointer variables of type T may refer to, These sets are called reference classes. They are not types. in Pascal (although the syntax for reference class appears in the early version of the Pascal specification [Wirth]). They are assertion language primitives and behave very much like unbounded arrays; their semantics are defined by axioms in Section 3.2.

DEFINITION (types)

i) INTEGER, REAL, and BOOLEAN are types.

ii) If T, TO, ... ,Tn are types and FO, . . . ,Fn are identifiers (field identifiers) then

ARRAY[K..L] OF T,

RECORD F0:T0; F1:T1; ...; Fn:Tn END,

↑T, and

P*T

are types.

iii) They are the only types.

In the definitions below. we use the following notation:

D,D'-- data structure representations,

C -- a reference class representation,

E -- a Pascal expression ,

I -- an integer type data structure representation,

N-- a type name,

Y -- a pointer type variable,

X--a pointer type data structure representation,

F--a field identifier,

S-- a selector sequence,

DEFINITION (selector sequences)

$$S ::= \phi \mid [I]\otimes S \mid \subset X \supset \otimes S \mid .F \otimes S$$

DEFINITION (S is applicable to D)

S is empty,

D is of type ARRAY[K..L] and S=[I]⊗S' and K≤I≤L and S' is applicable to D[I],

D is of type RECORD and S=.F⊗S' and F is a field of D and S' is applicable to D.F,

D is of type REFERENCE CLASS of N, and S=⊂X⊃⊗S'

and X is of type ↑N and S' is applicable to D⊂X⊃.

DEFINITION

(a) (reference class data structure representations)

C ::= P*N | CU(Y} | <C,S,D>

(b) (data structure representations)

D ::= E | C | <D,S,D'> | D⊗S

subject to the restrictions:

(i) S is applicable to C and D.

(ii) In <C,S,D> and <D,S,D'>,

type_of(C⊗S)=type_of(D) and type_of(D⊗S)=type_of(D').

This completes the definition of the syntax of data structure representations.

3.2 Axioms for data structure representations.

Ax 1. $D \otimes \phi = D$

Ax 2. $\langle D , \phi , E \rangle = E.$

Ax 3. $\langle D , [I] \otimes L \quad E \rangle [J] \otimes K =$
if i = J then $\langle D[I] , L , E \rangle \otimes K$ else $D \otimes [J] \otimes K.$

Ax 4. $\langle D , .F \otimes L , E \rangle \otimes .G \otimes K =$
if F = G then $\langle D \otimes .F , L , E \rangle \otimes K$ else $D \otimes .G \otimes K.$

Ax 5. $\langle D , cX \supset \otimes L , E \rangle \otimes cY \supset \otimes K =$
if X = Y then $\langle D \otimes cX \supset , L , E \rangle \otimes K$ else $D \otimes cY \supset \otimes K.$

Ax 6. $\langle D , L , D \otimes L \rangle = D.$

Ax' 7. $\langle\langle D , [I] \otimes L , V \rangle , [J] \otimes K , W \rangle =$
if I = J then $\langle D , [I] , \langle\langle D \otimes [I] , L , V \rangle , K , W \rangle\rangle$
else $\langle\langle D , [J] \otimes K , W \rangle , [I] \otimes L , V \rangle.$

Ax 8. $\langle\langle D , .F \otimes L , V \rangle , .G \otimes K , W \rangle =$
if F = G then $\langle D , .F , \langle\langle D \otimes .F , L , V \rangle , K , W \rangle\rangle$
else $\langle\langle D , .G \otimes K , W \rangle , .F \otimes L , V \rangle.$

Ax 9. $\langle\langle D , cX \supset \otimes L , V \rangle , cY \supset \otimes K , W \rangle =$
if X = Y then $\langle D , cX \supset , \langle\langle D \otimes cX \supset , L , V \rangle , K , W \rangle\rangle$
else $\langle\langle D , cY \supset \otimes K , W \rangle , cX \supset \otimes L , V \rangle.$

Ax 10. $D \cup \{X\} \otimes cY \supset \otimes K =$
if X = Y then Undefined else $D \otimes cY \supset \otimes K.$

Ax 11. if X ≠ Y then
$\langle D , cX \supset \otimes L , E \rangle \cup \{Y\} = \langle D \cup \{Y\} , cX \supset \otimes L , E \rangle$

Examples

We illustrate how properties of data structure representations can be proved using these axioms.

1) $I \neq J \supset \langle\langle A,[I],1\rangle,[J],2\rangle[I] = 1$

This statement says that after assigning 1 to the **I-th** element and 2 to the J-th element , the value of the **I-th** element is 1 if **I≠J.**

. Using Ax 3, the statement is reduced to

$I \neq J \supset \langle A,[I],1\rangle[I]=1.$

Then using Ax 3 again, it becomes

$I \neq J \supset 1=1.$

2) <<A,[I][J],2>,[K],B>[I][L]

= if K=I then

(if L=J then 2 else A[I][L]) else B[I][L]

Applying Ax 3 to the left-hand side of the equation reduces it to

if K=I then

<<A,[I][J],2>[I],ϕ,B>[L] else <A,[I][J],2>[I][L].

Applying Ax 2 to the then-part and Ax 3 to the else-part, we get

if K=I then B[L] else <A[I],[J],2>[L].

This finally reduces by Ax 3 to

if K=I then B[L] else if J=L then 2 else A[I][L].

3.3 Axioms for assignment and storage allocation.

Rule 1(Introduction of Reference Class Identifiers)

In all Boolean formulas, all dereferenced pointers, **X↑** , are replaced by **P*T⊂X⊃** where **type_of(X)=↑T.**

Examples:

```
X↑ → P#T⊂X⊃   assuming type-of (X)=T.
X↑.F → P#T⊂X⊃.F
A [X↑.F] → A [P#T⊂X⊃.F]
X↑.F↑.G → P#S⊂P#T⊂X⊃.F⊃.G   assuming type-of (X↑.F)=S.
                              Note that the introduction
                              must take place from inside out.
```

The reference class introduction rule can be formally defined by the following function ar. (ar stands for actual representation.)

- **ar(V)** = V ; if V is a simple variable

ar(A[I])= ar(A)[ar(I)] ;

ar(R.F) = ar(R).F ;

ar(Z↑) = P*T⊂ar(Z)⊃ ; where **type_of(Z↑)=T.**

Rule 2(General rule for assignment).

$$P \Big|^{arn\ (V)}_{\langle arn\ (V),\ are\ (V),\ E\rangle} \{\ V \leftarrow E\ \}\ P$$

where **arn(V)** is the name part of the actual representation of V and **ars(V)** is the selector sequence part of V. Thus, **ar(V)=arn(V)⊗ars(V).**

We can define **arn(V) and ars(V)** formally as follows.

arn(V) = V ; if V is a simple variable

arn(A[I])= arn(A) ;

arn(R.F) = arn(R) ;

arn(Z↑) = P*T ; where **type_of(Z↑)=T.**

ars(V) = ϕ ;

ars(A[I])= ars(A)⊗[ar(I)] ;

ars(R.F) = ars(R)⊗.F ;

ars(Z↑) = ⊂ar(Z)⊃ .

Rule 2 reduces in simple cases to rules in **[Hoare &** Wirth]:

1) Simple variable **V.**

In this case **arn(V)** = V and **ars(V) = ϕ**

So the rule **becomes**

$$P \Big|^{V}_{<V,\phi,E>} \{ V \leftarrow E \}\ P.$$

However, from Ax 2, **<V,ϕ,E>** = E. Thus, we obtain the original rule.

2) Simple 'array **V=A[I].**

arn(V)=A and ars(V)=[I]. So the simple array assignment rule is obtained **from** the general rule.

$$P \Big|^{A}_{<A,[I],E>} (A[I] \leftarrow E)\ P.$$

3) Simple record **V=R.F**

Then **arn(V)=R** and **ars(V)=.F.** So the simple **record** assignment rule is obtained from the general **rule.**

$$P \Big|^{R}_{<R,.F,E>} \{ R.F \leftarrow E \} \; P.$$

Rule 3.(Storage allocation)

$$\bigwedge_{V \in F} V \neq X' \supset Q \Big|^{P\#T}_{P\#T \cup \{X'\}} \Big|^{X}_{X'} \{ New(X) \} \; Q$$

whdre **type_of(X)=↑T.** X' is a newly created variable which does not appear anywhere, and F is the set of variables of Q whose types are **↑T.**

The allocation rule NEWB (Section 2.6) cannot be derived from Rule 3. NEWB is not suitable for implementation because of the potentially large number of terms in the SET-OF **P*T** each **of** which contributes an inequality in the premiss. This leads to very large Verification Conditions with large numbers of irreievent inequalities. The set F in Rule 3 is a "first approximation" to SET-OF **P*T.** The union notation for the extension of the reference class **P*T** permits the user to add documentation statements which have the effect of adding extra assumptions to the premiss.

For example, suppose we introduce a predicate **NOTEQUAL(C,D,D')** satisfying:

i. **NOTEQUAL(C,E,F)↔E≠F** for ail reference classes C and terms E **and** F,

ii. **NOTEQUAL(P*TU{X'},Y↑.S,X')** for ail variables Y and selector sequences S,

X' being the newly created variable,'

iii. NOTEQUAL(P*TU{ X'},Y,X') for ail variables Y different from X'.

Then we will be able to prove TRUE {NEW(Z)] Z≠X↑.CDR , This is not provable using Rule 3 alone although it is a consequence of NEWB.

4. EXAMPLES.

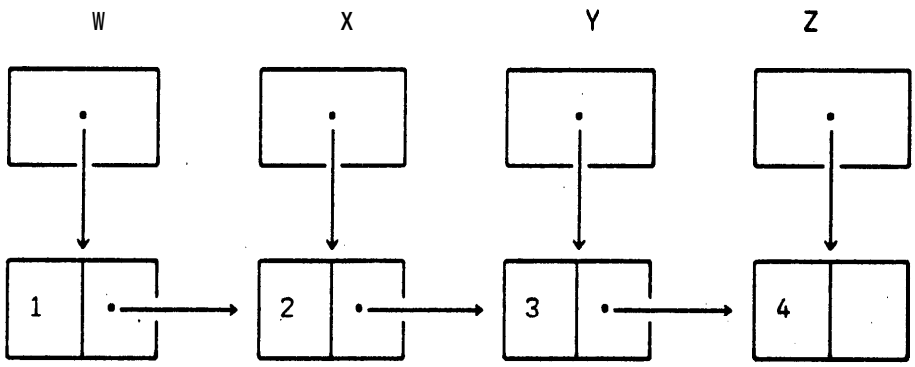
The extensions to the assertion language and proof rules defined in Section 3 have been implemented in the Stanford Pascal verifier. The verifier also uses axioms Axl-Ax6 (Section 3.2) to simplify VC's.

Some example verifications of programs with pointer type parameters are given below. Details of the verifier and studies of other applications can be found in [Suzuki a,b], [v.Henke & Luckham], and [Luckham & Suzuki]. In particular a methodology for verifying programs with this sort of verifier is outlined in [v.Henke & Luckham].

4.1 Side effects in pointer data structures.

Example 1. .

```
TYPE LINEAR-RECORD VAL:INTEGER; NEXT:↑LINEAR END:
VAR W,X,Y,Z:↑LINEAR;
BEGIN
        NEW(W);NEW(X);NEW(Y);NEW(Z);
        W↑.VAL  := 1;
        W↑.NEXT := X;
        X↑.VAL := 2;
        X↑.NEXT := Y;
        Y↑.VAL := 3;
        Y↑.NEXT := Z;
        Z↑.VAL := 4;
{At this point there is a four cell linear list. Fig. 1}
        X↑.NEXT := Z;
{Now, Y↑ has been cut out of the linear list. Fig.21
ASSERT  W↑. NEXT?, NEXT?. VAL-4
END.  .
```



Fig, 1

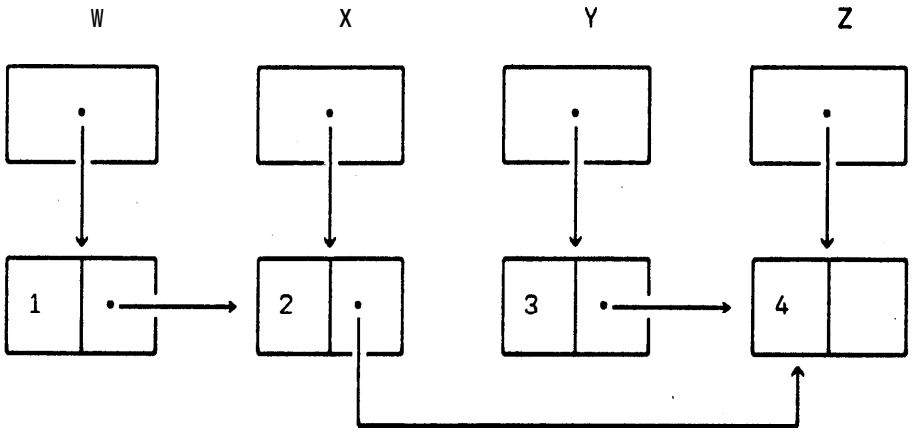


Fig.2

Fig. 2 shows the final state of the reference class P*LINEAR. The only operation involving W↑.NEXT↑.NEXT↑.VAL assigns 3 to the cell. That cell is then "short circuited" out of the list by an operation that does not explicitly mention it.

The result of giving example 1 to the verifier is a single VC; before simplification it looks like this:

*****

```
FOR THE MAIN PROGRAM
THERE ARE 1 VERIFICATION CONDITIONS


# 1
(¬Y00=Z00 & .
 ¬X00=Z00 &
 ¬W00=Z00 &
 ¬W00=Y00 &
 ¬X00=Y00 &
 ¬W00=X00 8
 TRUE
→
<<<<<<<<P#LINEARU {W00} U {X00} U {Y00} U {Z00} ,cW00⊃.VAL,1>,cW00⊃.NEXT,X00>,
cX00⊃.VAL,2>,cX00⊃.NEXT,Y00>,cY00⊃.VAL,3>,cY00⊃.NEXT,Z00>,cZ00⊃.VAL,4>,
cX00⊃.NEXT,Z00><<<<<<<<<P#LINEARU {W00} U {X00} U {Y00} U {Z00} ,cW00⊃.VAL,1>,
cW00⊃.NEXT,X00>,cX00⊃.VAL,2>,cX00⊃.NEXT,Y00>,cY00⊃.VAL,3>,cY00⊃.NEXT,Z00>,
cZ00⊃.VAL,4>,cX00⊃.NEXT,Z00><<<<<<<<<P#LINEARU {W00} U {X00} U {Y00} U {Z00} ,
cW00⊃.VAL,1>,cW00⊃.NEXT,X00>,cX00⊃.VAL,2>,cX00⊃.NEXT,Y00>,cY00⊃.VAL,3>,
cY00⊃.NEXT,Z00>,cZ00⊃.VAL,4>,cX00⊃.NEXT,Z00>cW00⊃.NEXT⊃.NEXT⊃.VAL=4)

 AFTER SORE SIMPLIFICATION, YOU CAN GET

# 1

TRUE

TIME: 21 CPU SECS, 54 REAL SECS
*****
```

The unsimplified VC has the form $Q \rightarrow (D \otimes S = 4)$ where D represents all the changes made to **P*LINEAR** (in order), and S selects **W↑.NEXT↑.NEXT↑.VAL.** (Clearly it would be nice to have a picture of D such as Fig. 2!) Variables X00, YOO, etc. and the inequalities between them result from the allocation rule.

In this example the simplification axioms (Section 3.2) reduce the VC completely to TRUE and no additional information is required of the user.

4.2 Verification Bases.

Verifications normally depend on user-supplied lemmas. The verifier uses these lemmas to simplify and prove VC's. If all VC's are reduced to TRUE this means that there is a proof that the program satisfies its ENTRY/EXIT specifications assuming the lemmas. The set of lemmas is called a BASIS of the verification. A basis is not necessarily a complete axiomatization of given programming concepts but need be only a set of lemmas provable from such an axiomatization. Indeed, the verifier can be viewed as an instrument for searching for reasonable sets of assumptions that imply the consistency of a program with its specifications. Methods for constructing and analysing bases are described in [v.Henke & Luckham].

Lemmas are stated in simple logical forms called AXIOMS and GOALS. They contain information about how they are to be used in proof searches; this need not concern us here. To read the lemmas as logical statements, simply ignore all "@" signs in the examples. Then a lemma of the form AXIOM A↔B is the logical equivalence A↔B, and COAL A SUB B is the implication B→A.

The following examples deal with verifying that programs maintain the loopfreeness of the list structures they operate on. The examples also show (a) the use of the extended assertion language to express concepts such as loopfreeness of lists, and (b) the characterization of concepts by lemmas in the basis.

4.3 Reachability in Linear Lists.

We -wish to verify the* loopfreeness of linear lists, in which each cell has one

pointer field, the NEXT field, which points to the next cell in the list. One way to approach this 'problem is to introduce a predicate **Reach(D,X,Y),** where D is a reference class representation of type reference class of T, and **X,Y** are both pointer variables of type **↑T. REACH(D,X,Y)** means that the sequence X, **X↑.NEXT, X↑.NEXT↑.NEXT,...** in the reference class D contains (or reaches) **Y.** This implies that the list structure between X and Y in D is **loopfree** under the NEXT operation, Notice that NEXT ought to **be** an **explicit** parameter of REACH, but since we are assuming that our list structures have only one NEXT field, we have omitted it.

Example 2 is the insertion of an element into the middle of a linear list. We verify that **Reach(D,ROOT,SENTINEL)** is still preserved after the insertion, ROOT and SENTINEL being pointers to the beginning and end of the list. **SENTINEL↑.NEXT=NILL** means that SENTINEL **points to the last** element of the list.

Example 2.

```
ENTRY REACH(P#WORD,ROOT,SENTINEL)∧ (Y≠SENTINEL)∧(SENTINEL↑.NEXT=NILL)∧
REACH (P#WORD, ROOT, Y ) ∧REACH (P#WORD, Y, SENT I NEL );

EXIT REACH(P#WORD,ROOT,SENTINEL);

TYPE REF =↑WORD;
TYPE WORD = RECORO COUNT: INTEGER; NEXT: REF END;

VAR Y,Z,ROOT,SENTINEL:REF;

BEGİN
        NEW(Z) ;
        Z↑.NEXT←Y↑.NEXT;
        Y↑.NEXT←Z;
END .;
```

The set of lemmas in the **goalfile** below' is a Basis for verifying example 2. We do not claim that it is a complete characterization of **REACH(D,X,Y),** but merely that each of the lemmas is an obvious property of REACH that would be provable **given** a complete set of axioms,

Thus Goal 1 states that for W to be reachable from X in a reference class resulting from class D by performing **Y↑.NEXT←Z,** it is sufficient that **REACH(D,X,Y)** and **REACH(D,Z,W)** and also **¬REACH(D,Z,Y)** to ensure that no loop is introduced by the operation. Clearly the truth of this lemma depends on more atomic properties e.g. **REACH(D,Y,Y↑.NEXT),** transitivity (Coal **4), and REACH(D,Y,Y)** (from which **¬REACH(D,Z,Y)** implies **Z≠Y).**

Goal 2 is a statement about a "short circuit" operation; **<D, ⊂Z⊃.NEXT, D⊂Y⊃.NEXT>** represents the reference class that results from D by **Z↑.NEXT←Y↑.NEXT.** This excludes Y from the sequence Z, **Z↑.NEXT,** . . . provided **Y≠Z** and Y cannot be reached from **Y↑.NEXT.** A loop might however, be introduced into the new structure unless **¬REACH(D,Y,Z).**

Coal 3 states sufficient conditions for Y not to be reachable from **Y↑.NEXT.**

Coal **5** is a typical frame axiom for storage allocation. It means that reachability is not affected by the allocation of a new cell; Goals 6 and **7** are similar.

Coals 8 and 9 state conditions for Reachability when operations are performed on a new cell.

It turns out that only goals **1,2,3,6,8,9** are used in proving the verification condition below.

```
GOALFILE

G1: GOAL REACH(<@D,c@Yɔ.NEXT,@Z>,@X,@W)
     SUB REACH(D,X,Y)∧¬REACH(D,Z,Y)∧REACH(D,Z,W);

G2: GOAL ¬REACH(<@D,c@Zɔ.NEXT,@Dc@Yɔ.NEXT>, @Z, @Y)
     SUB (Z≠Y)∧ ¬REACH(D,DcYɔ.NEXT,Y);

G3: GOAL ¬REACH(@D,@Dc@Yɔ.NEXT,@Y)
     SUB (NILL=Dc@Sɔ.NEXT)∧ REACH(D,Y,@S);

G4: GOAL REACH(@D,@X,@Y)
     SUB REACH(D,X,@Z)∧REACH(D,@Z,Y);

G5: GOAL REACH(@Du{@Z}, @X,@Y)
     SUB REACH(D,X,Y)∧(Z≠X)∧(Z≠Y);

G6: GOAL ¬REACH(@Du{@Z}, @Du{@Z}c@Xɔ.NEXT,@Y)
     SUB ¬REACH(D,DcXɔ.NEXT,Y);

G7: GOAL (@Du{@Z}c@Sɔ=NILL)
     SUB (DcSɔ=NILL);

G8: GOAL REACH(<@Du{@Z},c@Zɔ.NEXT,@W>,@X,@Y)
     SUB REACH(D,X,Y)∧(Z≠X)∧(Z≠Y);

G9: GOAL REACH(<@Du{@Z},c@Zɔ.NEXT,@Du{@Z}c@Yɔ.NEXT>,@Z,@W)
     SUB  REACH(D,Y,W)∧(Z≠Y);
```

The result of giving the verifier the **goalfile** and example 2 is the following:

```
*****

FOR THE MAIN PROGRAM
THERE ARE 1 VERIFICATION CONDITIONS

#1
(¬SENTINEL=Z00 &
 ¬ROOT=Z00 &
 ¬Y=Z00 &
 REACH(P#WORD,ROOT,SENTINEL) &
-Y-SENTINEL &
 P#WORDcSENTINELɔ.NEXT=NILL &
 REACH(P#WORD,ROOT,Y) 6
 REACH(P#WORD,Y,SENTINEL)
→
 REACH(<<P#WORDu{Z00},cZ00ɔ.NEXT,P#WORDu{Z00}cYɔ.NEXT>,cYɔ.NEXT,Z00>,ROOT,
 SENTINEL))
```

```
AFTER SORE SIMPLIFICATION, YOU CAN GET

# 1    TRUE

*****
```

Notice that the reference class expression in the unsimplified VC conclusion represents the result of executing example 2. So this VC might itself be accepted as a lemma about insertion operations in the verification of more complex programs.

Example 3 illustrates what happens when we reverse the order of instructions in the example 2. The program is no longer correct in that it does introduce a loop into a **loopfree** structure. The program was run through the verifier with the same **GOALFILE** that was used previously.

Example 3.

```
ENTRY REACH(P#WORD,ROOT,SENTINEL)∧ (Y≠SENTINEL)∧(SENTINEL↑.NEXT=NILL)∧
REACH(P#WORD,ROOT,Y)∧REACH(P#WORD,Y,SENTINEL);

EXIT REACH(P#WORD,ROOT,SENTINEL);

TYPE REF =↑WORD;
TYPE WORO = RECORO COUNT:INTEGER; NEXT: REF END:

VAR Y,Z,ROOT,SENTINEL:REF;

BEGIN
        NEW(Z);
        Y↑.NEXT←Z;
        Z↑.NEXT←Y↑.NEXT;
END .;
```

```
*****

FOR THE MAIN PROGRAM
THERE ARE 1 VERIFICATION CONDITIONS

# 1
(¬SENTINEL=Z00 &
-ROOT-Z00 &
 ¬Y=Z00 &
 REACH(P#WORD,ROOT,SENTINEL) &
 ¬Y=SENTINEL 8
 P#WORD<SENTINEL>.NEXT=NIL &
 REACH(P#WORD,ROOT,Y) &
 REACH(P#WORD,Y,SENTINEL)
→
 REACH(<<P#WORD∪{Z00},<Y>.NEXT,Z00>,<Z00>.NEXT,<P#WORD∪{Z00},<Y>.NEXT,Z0~
0><Y>.NEXT>,ROOT,SENTINEL))

 AFTER SOME SIMPLIFICATION, YOU CAN GET

# 1

(¬Z00=Y &
 REACH(P#WORD,ROOT,SENTINEL) &
 ¬Y=SENTINEL &
 P#WORD<SENTINEL>.NEXT=NIL &
 REACH(P#WORD,ROOT,Y) &
 REACH(P#WORD,Y,SENTINEL) &
¬Z00=SENTINEL &
 ¬Z00=ROOT
→
 REACH(<<P#WORD∪{Z00},<Y>.NEXT,Z00>,<Z00>.NEXT,Z00>,ROOT,SENTINEL))

*****
```

The loop construction can be seen by analysis of the reference class expression in the conclusion of the simplified VC. The simplification results from Axioms 3.2. It is now easy to see that the final operation represented is **Z↑.NEXT←Z** which clearly introduces a loop.

4.4 Root and Sentinel Problem

This program was suggested by N. Wirth. It operates on a linear list. Each cell of the list has three fields: KEY, COUNT, and NEXT. KEY field contains the identification name for the cell, COUNT field contains the number of times SEARCH is called with the corresponding KEY, and NEXT field contains the pointer to the next cell in the list. ROOT points to the first cell and SENTINEL points to the next to the last cell. The last cell a dummy cell.

```
TYPE REF=↑WORD;
TYPE WORD=RECORD KEY:INTEGER;COUNT: INTEGER;NEXT:REF END;
VAR K:INTEGER;
ROOT, SENTINEL:REF;

PROCEDURE SEARCH(X:INTEGER;SENTINEL:REF;VAR ROOT:REF);

VAR W1,W2:REF;
BEGIN   W1←ROOT;
        SENTINEL↑.KEY←X;
        IF W1-SENTINEL THEN
        BEGIN
                NEW(ROOT);
                ROOT↑.KEY←X;  ROOT↑.COUNT←1; ROOT↑.NEXT←SENTINEL;
        END ELSE
        IF W1↑.KEY =X THEN W1↑.COUNT←W1↑.COUNT+1 ELSE
        BEGIN
                REPEAT  W2←W1; W1←W2↑.NEXT
                        UNTIL W1↑.KEY=X;
                IF W1-SENTINEL THEN
                BEGIN
                        W2←ROOT; NEW(ROOT);
                        ROOT↑.KEY←X; ROOT↑.COUNT←1; ROOT↑.NEXT←W2
                END ELSE
                BEGIN
                        W1↑.COUNT←W1↑.COUNT+1;
                        W2↑.NEXT←W1↑.NEXT;
                        W1↑.NEXT←ROOT; ROOT←W1
                END
        END
END:,,
```

In order 'to verify this program we have to show that several properties hold. Here are some of them. (1) The list structure is always loopfree and SENTINEL is reachable from ROOT. (2) If a cell with the given KEY exists in the list, no new cell is added; otherwise, one cell is added. (3) No two KEY's of cells in the list are the same. (4) After execution the list is reordered so that the first cell has the same KEY as the given KEY argument of SEARCH, and the order of the other cells is unchanged. (5) Only the COUNT field of the cell with the given KEY is incremented by 1, and the rest are unchanged. And finally the program terminates, Here we are going to show a verification that the first two properties -- reachability and non deletion -- hold.

Example 4 is the program with assertions about reachability. The ENTRY and EXIT assertions state that loopfreeness is maintained. The only additional documentation is an invariant describing obvious properties of the variables in the REPEAT loop.

Example 4.

```
PASCAL
TYPE REF=↑WORD;
TYPE WORD=RECORD KEY:INTEGER;COUNT:INTEGER;NEXT:REF END:
VAR K:INTEGER;
ROOT, SENTINEL:REF;

PROCEDURE SEARCH(X: I NTEGER;SENTINEL:REF;VAR ROOT:REF);
ENTRY REACH(P#WORD,ROOT,SENTINEL)∧(SENTINEL↑.NEXT=NILL);
EXIT REACH(P#WORD,ROOT,SENTINEL);

VAR W1,W2:REF;
BEGIN   W1←ROOT;
        SENTINEL↑.KEY←X;
        IF W1=SENTINEL THEN
        BEGIN
                NEW(ROOT);
                ROOT↑.KEY←X; ROOT↑.COUNT←1; ROOT↑.NEXT←SENTINEL;
        END ELSE
        IF W1↑.KEY =X THEN W1↑.COUNT←W1↑.COUNT+1 ELSE
        BEGIN
                REPEAT W2←W1; W1←W2↑.NEXT
                        INVARIANT
                        REACH(P#WORD,ROOT,W2)∧(W1=W2↑.NEXT)∧(W2=SENTINEL)∧
                        REACH(P#WORD,W1,SENTINEL)∧(SENTINEL↑.KEY=X)∧
                        (SENTINEL↑.NEXT=NILL)
                        UNTIL W1↑.KEY=X;
                IF W1-SENTINEL THEN
                BEGIN
                        W2←ROOT; NEW(ROOT);
                        ROOT↑.KEY←X; ROOT↑.COUNT←1; ROOT↑.NEXT←W2
                END ELSE
                BEGIN
                        W1↑.COUNT←W1↑.COUNT+1;
                        W2↑.NEXT←W1↑.NEXT;
                        W1↑.NEXT←ROOT; ROOT←W1
                END
        END
END:.:
```

Below is a **GOALFILE** containing a basis that is sufficient to verify Example 4 (i.e. that the program satisfies its documentation). Comments explaining some of the goals appear between % signs. It turned out that goals 9,12, were not used in this verification.

```
GOALFILE

G1:   AXIOM REACH(@D,@X,@X) ↔ TRUE;

G2:   GOAL REACH(@D,@X,@Y)
      . SUB REACH(D,X,@Z)∧REACH (D,@Z,Y);

G3:   GOAL REACH(@D,@R,@Dc@X⊃.NEXT) SUB REACH(D,R,X);

G4:   GOAL REACH(@D,@Dc@X⊃.NEXT,@Y) SUB ¬(X=Y)∧REACH(D,X,Y);
   %X↑.NEXT is between X and Y%

G5:   GOAL ¬(@X=@Y) SUB ¬(@DcX⊃.KEY = @DcY⊃.KEY);
   %KEY fields of distinct cells are distinct%

G6:   GOAL ¬(@W=@Dc@Y⊃.NEXT) SUB
           ¬REACH(D,DcY⊃.NEXT,W);
%This is a special case of: if W is not reachable from
 X then X≠W.% .

G7:   AXIOM REACH(<@D,c@X⊃.KEY,@E>,@Y,@Z) ↔ REACH(D,Y,Z);

G8:   AXIOM REACH(<@D,c@X⊃.COUNT,@E>,@Y,@Z) ↔ REACH(D,Y,Z);

  %AXIOMS 7 and 8 state that operations on the KEY and COUNT fields
   do not alter loopfreeness%

G9:   GOAL ¬REACH(@DU{@X},@X,@Z) SUB ¬(X=Z);

G10: GOAL ¬REACH(@DU{@X},@Z,@X) SUB ¬(X=Z);

G11:  GOAL REACH(@DU{@Z},@X,@Y)
      SUB ¬(Z=X)∧ ¬(Z=Y)∧REACH(D,X,Y);
   %3-11 define the Reachability relation on newly allocated cells%

G12: GOAL REACH(<@DU{@Z},c@Z⊃.NEXT,@DU{@Z}c@Y⊃.NEXT>,@Z,@W)
      SUB ¬(Z=Y)∧REACH(D,Y,W);

G13: -GOAL REACH(<@D,c@Y⊃.NEXT,@Z>,@X,@W)
      SUB REACH(D,X,Y)∧¬REACH(D,Z,Y)∧REACH(D,Z,W);
   %12,13 describe sufficient conditions for preservation of
    Reachability when Z is inserted by operations similar
```

to example 2%

G14: GOAL REACH(<@D,c@Y⊃.NEXT,@Z>,@X,@W)
SUB REACH(D,X,Y)∧REACH(D,Y,Z)∧REACH(D,Z,W)∧¬(Y=Z);
%14 gives sufficient conditions for preservation of Reachability when cells between Y and Z are cut out of the list%

G15: GOAL -REACH (<@D,c@Y⊃. NEXT.@Z>, @X, @W) SUB
REACH(D,X,Y)∧REACH(D,Y,W)∧REACH(D,W,Z)∧¬REACH(D,Z,W)∧
¬(Y=W)∧¬(W=Z);
%15 states that if W is strictly between Y and Z, and there are no loops back to W after Z, then W cannot be reached after cutting out the cells between Y and Z.%

G16: GOAL -REACH (@D, @Dc@X⊃. NEXT, @Y)
SUB REACH(D,Y,X)∧REACH(D,X,@S)∧(Dc@S⊃.NEXT=NILL);
%Y cannot be reached from X↑.NEXT if X can be reached from Y and there are no loops after X. Here S is the end cell of the list structure and if it is reachable from X then there are no loops after X.%

Below is the annotated program to prove the subset property, i.e. the cells of the input **list** are a subset of those of the output. We have introduced a function **LIST(X,Y,D)** which is defined if **REACH(D,X,Y)** and whose value is the set of cells between pointers **X** and Y excluding **Y↑** in reference class **D.** Also we use the predicate **SUBSET(A,B).**

Example **5.**

```
PASCAL
TYPE REF=↑WORD;
TYPE WORD=RECORD KEY:INTEGER;COUNT:INTEGER;NEXT:REF END;
VAR K:INTEGER;
ROOT, SENTINEL:REF;

PROCEDURE SEARCH(X:INTEGER;SENTINEL:REF;VAR ROOT:REF);
ENTRY (P#WORD=P0)A(ROOT=R0)∧REACH (P#WORD,ROOT,SENTINEL)A
      (SENTINEL↑.NEXT=NILL);
EXIT SUBSET(LIST(R0,SENTINEL,P0),LIST(ROOT,SENTINEL,P#WORD));
VAR W1,W2:REF;
BEGIN   W1←ROOT;
        SENTINEL↑.KEY←X;
        IF W1-SENTINEL THEN
        BEGIN
                NEW(ROOT);
                ROOT↑.KEY←X; ROOT↑.COUNT←1; ROOT↑.NEXT←SENTINEL;
        END ELSE
        IF W1↑.KEY=X THEN W1↑.COUNT←W1↑.COUNT+1 ELSE
        BEGIN
                REPEAT W2←W1; W1←W2↑.NEXT
                INVARIANT
                SUBSET(LIST(R0,SENTINEL,P0),LIST(ROOT,SENTINEL,P#WORD))
                ∧(SENTINEL↑.KEY=X)∧(SENTINEL↑.NEXT=NILL)
                ∧REACH(P#WORD,ROOT,W2)∧REACH(P#WORD,W1,SENTINEL)
                ∧(<P0,⊂SENTINEL⊃.KEY,X>=P#WORD)
                ∧(W1=W2↑.NEXT)∧(W2≠SENTINEL)
                UNTIL W1↑.KEY=X;
                IF W1-SENTINEL THEN
                BEGIN
                        W2←ROOT; NEW(ROOT);
                        ROOT↑.KEY←X; ROOT↑.COUNT←1; ROOT↑.NEXT←W2   .
                END ELSE
                BEGIN
                        W1↑.COUNT←W1↑.COUNT+1;
                        W2↑.NEXT←W1↑.NEXT;
                        W1↑.NEXT←ROOT; ROOT←W1
                END
        END
END,.,
```

This **GOALFILE** together **with the previous GOALFILE** for reachability form a Basis for verifying Example S. **The AXIOMS here describe straightforward** properties of LIST **and** SUBSET. **UNION is the usual union operation** on sets.

```
GOALFILE

1.  AXIOM LIST(@X,@Y,<@D,c@K>.KEY,@Z>) ↔ LIST(X,Y,D);
2.  AXIOM LIST(@X,@Y,<@D,c@K>.COUNT,@Z>) ↔ LIST(X,Y,D);
3.  AXIOM IF (X≠Z)∧(Y≠Z) THEN LIST(@X,@Y,@D∪{@Z}) ↔ LIST(X,Y,D);
4.  AXIOM, IF REACH(D,R0,X)∧REACH(D,Y,R1)∧¬REACH(D,Y,X)
        THEN LIST(@R0,@R1,<@D,c@X>.NEXT,@Y>)
            ↔ UNION(LIST(R0,DcX>.NEXT,D),LIST(Y,R1,D));
5.  AXIOM IF REACH(D,Z,X)∧ ¬REACH(D,X,Z)
        THEN LIST(@X,@Y,<@D,c@Z>.NEXT,@E>)↔LIST(X,Y,D);
6.  AXIOM LIST(@R,@R,@D) ↔ ZERO;

7.  AXIOM UNION(@D,ZERO)↔D;
8.  AXIOM UNION(LIST(@X,@Y,@D),
            UNION(LIST(@R,@X,@D),LIST(@Y,@S,@D)))
        ↔ LIST(R,S,D);

9.  AXIOM SUBSET(@X,@X) ↔ TRUE:
10. AXIOM SUBSET(ZERO,@X) ↔ TRUE;
11. AXIOM SUBSET(@X,UNION(@Y,@X)) ↔ TRUE;

.;
```

Acknowledgement

We, ‘wish to thank our colleagues Derek Oppen and Robert Cartwright for many debates and discussions based on early drafts of this paper which were very helpful and resulted in definite improvements.

Bibliography

[Burstall] Burstall, R. M.,
Some Techniques for Proving Correctness of Programs which Alter Data Structures,
Machine Intelligence 7,
Edinburgh University Press,
Nov. 1972.

[von Henke & Luckham] von Henke, F. W. and D. C. Luckham,
A Methodology for Verifying Programs,
Proceedings of International Conference of Reliable Software,
IEEE, pp.156-164, 1975.

[Hoare 69] Hoare, C. A. R.,
An Axiomatic Basis for Computer Programming,
CACM, Vol. 12, 1969, Oct., pp.576-580.

[Hoare 71] Hoare, C. A. R.,
Procedures and Parameters: an axiomatic approach,
Symposium on Semantics of Algorithmic Languages,
E. Engeler(ed.), Springer-Verlag, 1971, pp.102-116.

[Hoare & Wirth] Hoare, C. A. R. and N. Wirth,
An Axiomatic Definition of the Programming Language PASCAL,
Acta Informatic, Vol. 2, 1973, pp.33593SS.

[ILL] Igarashi, S. and R. L. London, and D. C. Luckham,
Automatic Program Verification I: Logical Basis and Its Implementation,
Acta Informatica, Vol. 4, pp.145-182, 1975.

[King] King, J. C.
.A *Program Verifier,*
Ph.D. thesis, Carnegie-Mellon University, 1969.

[McCarthy] McCarthy, J.,
A Formal Description of a Subset of ALGOL,

Formal Language Description Languages **for Computer Programming,** **Proc. IFIP** Working Conference **1964(T. B.** Steel, Jr. **ed.),pp.1-12,** North-Holland Publishing Co., Amsterdam, **1966.**

[Luckham & Suzuki] **Luckham,** DC. and **N.** Suzuki,
Automatic. Program Verification IV:
Proof of Termination within Weak Logic of Programs,
Stanfor Artificial Intelligence Laboratory Memo 269,
October, 1975.

[Oppen & Cook] 0ppen, DC. and **S.A.** Cook,
Proving Assertions about Programs that Manipulate Data Structures,
Proc. of 7th Annual ACM Symp. **on** Theory **of Computing, May 1975.**

[Spi tzen **&** Wegbrei **t]**
***The** Verification and **Synthesis** of Data Structures,*
Acta Informatica, Vol. 4, No. **2, 1975, pp.127-144.**

[Suzuki a] Suzuki, Norihisa,
Verifying Programs by Algebraic and Logical Reduction,
Proceedings of **Intl. Conf.** on Reliable Software,
SICPLAN Notices, June **, 1975, pp.473-481.**

[Suzuki b] Suzuki, Norihisa,
Automatic Verification of Programs with Complex **Data Structure,**
Ph.D. Thesis, Stanford University, **1975.**

[Wirth71] Wirth, Niklaus,
The Programming Language Pascal,
Acta Informatica, Vol. 1, No. 1, 1971, pp.35-63.