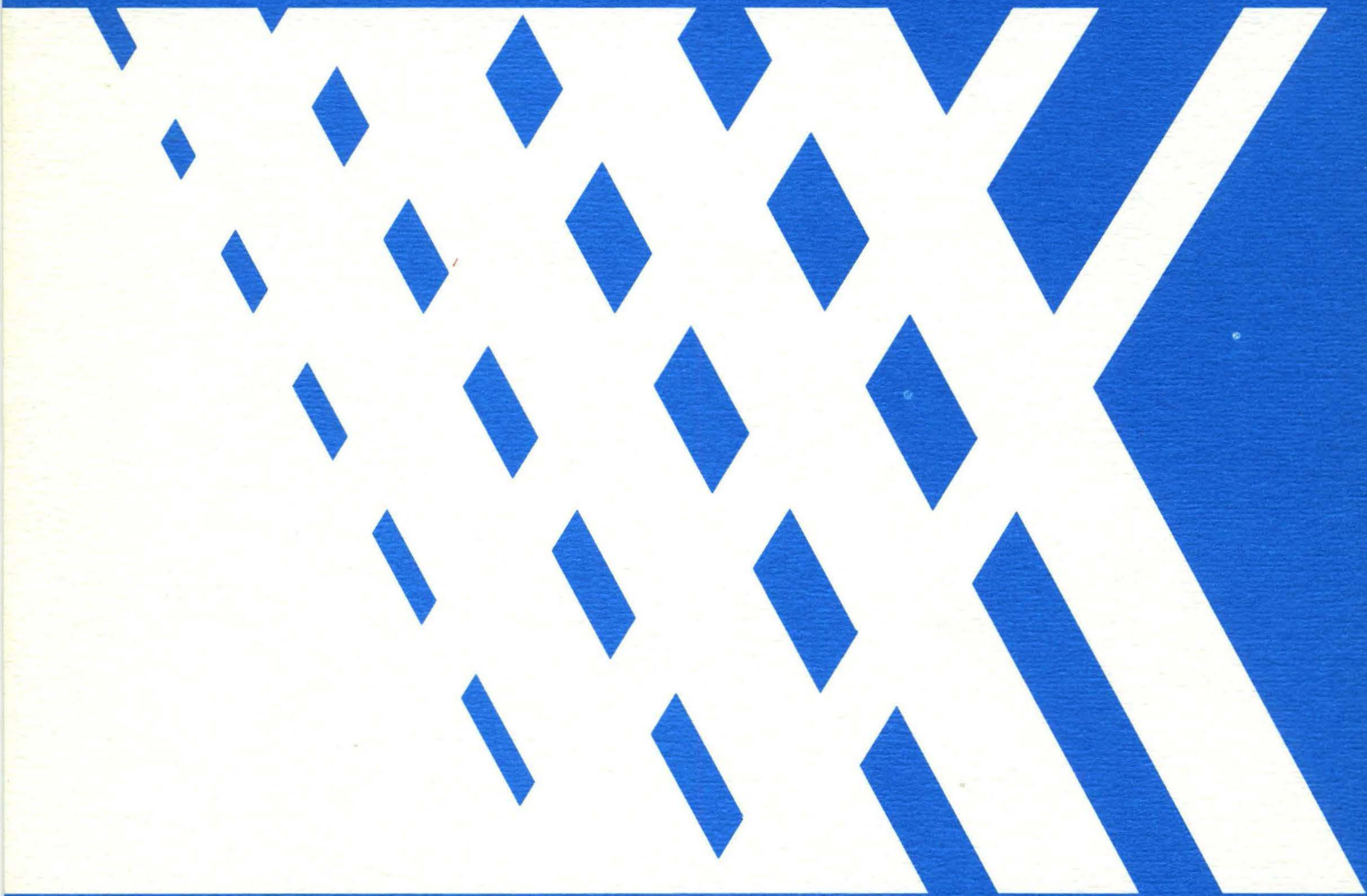
CONSTRUCTIVE METHODS IN PROGRAM VERIFICATION

BY BEN WEGBREIT



XEROX
PALO ALTO RESEARCH CENTER

CONSTRUCTIVE METHODS IN PROGRAM VERIFICATION

BY BEN WEGBREIT

CSL-76-2 JULY 1976

Most current approaches to mechanical program verification transform a program and its specifications into first order formulas and try to prove these formulas valid. Since the first order predicate calculus is not decidable, such approaches are inherently limited. This paper proposes an alternative approach to program verification: Correctness proofs are constructively established by *proof justifications* written in an algorithmic notation. These *proof justifications* are written as part of the program, along with the executable instructions and correctness specifications. A notation is presented in which instructions, specifications, and justifications are neatly interwoven. The justifications establish the connection between the instructions and specifications; they document the reasoning on which the correctness is based. Programs so written may be verified by proving the truth of quantifier-free logical formulas.

KEY WORDS

program verification, proving programs correct, proof justifications, constructive program verification

CR CATEGORIES

4.19, 4.22, 5.21, 5.24

XEROX

PALO ALTO RESEARCH CENTER

3333 COYOTE HILL ROAD / PALO ALTO / CALIFORNIA 94304

1. INTRODUCTION

Most research in mechanical program verification has approached the problem by seeking ways in which to transform a program and its specifications into logical formulas and trying to prove these formulas valid. Pioneering studies developing this approach include [2], [4], [5], and [7]. The specification language is typically a first order theory. The formulas to be proved also belong to the theory. It is well known that there is no decision procedure for the first order predicate calculus. Thus, this approach is necessarily limited - a mechanical program verifier cannot guarantee that in finite time it will verify each correct program and reject each incorrect one.

These observations may be given substance by examining contemporary verification systems. The theorem provers employed tend to flounder on large or complex programs. Because of this, the most recent verification systems rely on assistance from the programmer, either as interactive guidance [1], [3], [10] or as a file of axioms to use in the proof [9].

Most verification systems use the method of inductive assertions [2], or some related induction rule [8]. The induction rule is used to form *verification conditions* as the formulas to be proved. Such induction rules create a second problem: Even if the program is correct with respect to its input-output specifications and all the internal specifications are correct, *the verification conditions may not be valid if the specifications are too weak* [11]. Writing specifications which are strong enough to make the verification succeed is therefore the responsibility of the programmer. Experience to date indicates this is difficult. We conjecture that the difficulty is partly caused by the absence of an explicit link between the specifications and executable instructions.

This paper proposes an alternative approach. If a correctness specification is difficult to establish, it is desirable to justify its proof. *We suggest that proof justifications should be written in an algorithmic notation and that these justifications should be written as part of their programs*, along with executable instructions and correctness specifications. We propose some notation for this purpose that we have found convenient to use and aesthetically pleasing. For example, in this notation, a **while** loop includes a *specification* and a *justification*, in addition to the usual loop test and body. The *specification* may be an inductive assertion; it documents the values computed. The *justification* establishes how the correctness proof is to proceed; it documents the connection between the instructions and the specification. Taking the *justification* as a directive, it is readily decided whether the instructions, specifications, and justifications agree. Just as with other language constructs, justifications often have a regular structure. Default rules are adopted so that the most common cases need not be written - an omitted justification invokes the default. The body of this paper amplifies this cryptic sketch.

This approach has a potential weakness, since it requires the programmer to write additional text. Our observations have indicated that the additional work is either small or well-spent: In simple cases, justifications can be defaulted. Where justifications cannot be defaulted, the clarity they add is ample recompense for the effort expended.

We believe that justifications are a useful tool in understanding programs - their design, refinement, and modification. Our thesis is prescriptive: *programs should contain their justifications.*

2. JUSTIFYING UNIVERSALLY QUANTIFIED ASSERTIONS

We introduce *proof justifications* with a small example. As inductive assertions are the most familiar kind of correctness specifications, we use them throughout; a similar development could be given for other kinds of specifications. The inductive assertion method reduces program correctness to correctness of a finite set of finite *paths*. A program *path* starts with an initial assertion, continues with executable instructions, and terminates with a final assertion. There are well-known algorithms for mapping a *path* into its verification condition.

We use several notations to write compact assertions: Closed intervals are written [i:f]; $k \in [i:f]$ means $i \leq k \leq f$; $(\forall k \in [i:f], P(k))$ means $(\forall k)\ i \leq k \leq f \rightarrow P(k)$. The pair {, } is used interchangably with **begin**, **end** as delimiters.

Consider the following *path*

assert $(\forall j \in [1:J-1],\ D[j] < A[I]) \ \wedge$
    $(\forall r,s \in [1:M],\ r \leq s \rightarrow D[r] \leq D[s]) \ \wedge$
    $(\forall i \in [1:I-1],\ \forall m \in [1:M],\ D[m] \neq A[i])$;
if $A[I] < D[J]$ **then** $I \leftarrow I+1$;
assert $(\forall i' \in [1:I-1],\ \forall m' \in [1:M],\ D[m'] \neq A[i'])$

The final assertion is a logical consequence of the initial assertion and the intervening statement. This may be demonstrated as follows:
(I) Suppose $A[I] \geq D[J]$, so the final value of I is the same as the initial value of I; then the final assertion is a consequence of the third initial assertion.
(II) Suppose that $A[I] < D[J]$, so the final value of I is the initial value of I plus 1; let I' denote the final value of I. Consider some fixed i' and m' such that $i' \in [1:I'-1]$ and $m' \in [1:M]$.
    II.A) If i' lies in the closed interval 1 to I'-2 then the final assertion is a consequence of the third initial assertion, choosing i and m to be i' and m' respectively.

II.B) If i' is I'-1 then there are two further cases:

II.B.1) If m' lies in the interval 1 to J-1, the first initial assertion can be used, taking j to be m'

II.B.2) If m' lies in the interval J to M, use the program test A[I]<D[J] and choose r and s of the second initial assumption to be J and m' respectively.

This demonstration is clearly sound. It may be stated in general terms as follows: To prove that a universally quantified assertion (∀k') P(k') is a logical consequence of a universally quantified assertion (∀k) Q(k), it suffices to show that for each value of k' there is some value of k such that P(k') follows from Q(k). If the assertion to be proved has the form (∀k'∈[i:f], P(k')), it suffices to show that for each k' in the interval [i:f] there is some choice for k which makes the formula true.

In the paragraph above, we described the choices in English. Alternatively, we might use an algorithmic notation to express our choices. If k and k' are quantified variables, we write *k←k'* to mean that *k' is fixed and k is chosen to be k'*. We call this an *instantiation* of k. Using this notation, the choices to establish the above proof can be written

if A[I]≥D[J] ∨ i'∈[1:I-2] **then** {i←i', m←m'}
elseif m'∈[1:J-1] **then** j←m'
else {r←J, s←m'}

This is a *proof justification.*

To make clear how the instantiations are used in the proof, it is sometimes helpful to display the body of the instantiated formula following the instantiation. For example, the instantiation r←J, s←m' may be expanded to r←J, s←m': D[J]≤D[m'], where the colon may be read as "so that". The instantiated formula D[J]≤D[m'] following the colon adds no new information. However, the programmer and subsequent readers may find it helpful. Its use is a matter of taste. Similarly, redundant tests may be added to make explicit the conditions for each *justification* case, e.g. **if** P **then** E **else** F may be expanded to **if** P **then** E **elseif** ~P **then** F. Using these expanded forms, the above *justification* may be written

if A[I]≥D[J] ∨ i'∈[1:I-2] **then** {i←i', m←m': D[m']≠A[i']}
elseif i'=I-1 ∧ m'∈[1:J-1] **then** j←m': D[m']<A[I'-1]
elseif i'=I-1 ∧ m'∈[J:M] **then** {r←J, s←m': D[J]≤D[m']}

An assertion may be annotated by using a *justification*, i.e.

assert <*assertion*> **using** <*justification*>

A *justification* which annotates an *assertion* is used to prove verifications conditions for *paths* which terminate with that *assertion.* Often, a *justification* instantiates quantified variables of the initial assertion of a path; to avoid name clashes, we adopt the convention that no quantified variable name is repeated in a procedure.

The justified program may be written

```
assert (∀j∈[1:J-1], D[j]<A[I]) ∧
    (∀r,s∈[1:M], r≤s→D[r]≤D[s]) ∧
    (∀i∈[1:I-1], ∀m∈[1:M], D[m]≠A[i]);
if A[I]<D[J] then I←I+1;
assert (∀i'∈[1:I-1], ∀m'∈[1:M], D[m']≠A[i'])
using if A[I]≥D[J] ∨ i'∈[1:I-2] then {i←i', m←m': D[m']≠A[i']}
    elseif i'=I-1 ∧ m'∈[1:J-1] then j←m': D[m']<A[I-1]
    elseif i'=I-1 ∧ m'∈[J:M] then {r←J, s←m': D[J]≤D[m']}
```

Given this program, one may check its correctness by considering each case specified by the justification. Each case is a quantifier-free formula which is simple to prove. To complete the proof, it suffices to check that the specified cases cover all ways in which the variables i' and m' can be instantiated. Verifying correctness mechanically is correspondingly simple.

As above, the cases of a justification typically correspond to conditions of the program. Because of this, it is often convenient to intermix the parts of a justification with the program conditions to which they correspond. We prefix such justifications with the keyword **use.** A justification appearing on a program path is to be used in proving verification conditions generated by that path. With these conventions, the program may be written

```
assert (∀j∈[1:J-1], D[j]<A[I]) ∧
    (∀r,s∈[1:M], r≤s→D[r]≤D[s]) ∧
    (∀i∈[1:I-1], ∀m∈[1:M], D[m]≠A[i]);
if A[I]<D[J] then
    begin I←I+1;
    use if i'∈[1:I-2] then {i←i', m←m': D[m']≠A[i']}
        elseif i'=I-1 ∧ m'∈[1:J-1] then j←m': D[m']<A[I-1]
        elseif i'=I-1 ∧ m'∈[J:M] then {r←J, s←m': D[J]≤D[m']}
    end
else use {i←i', m←m': D[m']≠A[i']};
assert (∀i'∈[1:I-1], ∀m'∈[1:M], D[m']≠A[i'])
```

The **using** clause is empty here. It's information has been distributed into the **if** statement. That statement has been changed from an **if-then**, to an **if-then-else**, where the **else** clause has no executable instructions - only a justification.

It should be stressed that the quantifier instantiations $i \leftarrow i'$ and $m \leftarrow m'$ have a rather different interpretation from the increment instruction $I \leftarrow I+1$. The instantiations $i \leftarrow i'$ and $m \leftarrow m'$ are *proof justifications.* They state that their path can be proved by choosing i and m in $(\forall i \in [1:I-1],\ \forall m \in [1:M],\ D[m] \neq A[i])$ to be i' and m', obtaining $(i' \in [1:I-1] \wedge m' \in [1:M] \rightarrow D[m] \neq A[i])$; no other use need be made of $(\forall i \in [1:I-1],\ \forall m \in [1:M],\ D[m] \neq A[i])$. Overloading the binary connective "←" with this new interpretation seems fitting, for it suggests a constructive approach to proofs - an approach we find well-suited to verifying programs.

Concerning mechanical verification, *we regard the justifications as essential, not merely helpful hints.* If needed justifications are missing or wrong, we require a verification system to report the inconsistency but we do not expect it to debug the justifications. In particular, this implies that such a verifier may reject an otherwise correct program if the justifications are wrong. We noted in the introduction that mechanical program verifiers already suffer from a form of incompleteness: if the assertions are too weak, a correct program with correct assertions will yield an invalid verification condition and may therefore be rejected. The present approach introduces a related incompleteness in regard to justification.

3. JUSTIFICATIONS

In the preceding example, a justification assigns values to quantified variables of the initial assertion, as functions of given values for quantified variables in the final assertion. This occurs because the quantified variables were bound in universally quantified formulas at the outermost level of both assertions. Had the variables been existentially quantified, the roles of final and initial assertions would have been exchanged: a justification would then assign values to quantified variables of the final assertion as a function of given values in the initial assertion.

For example, consider the following straight-line fragment taken from a search loop which finds a zero element in the array B

assert $(\exists e \in [I:N],\ B[e]=0)$;
if $B[I]=0$ **then** $N \leftarrow I$ **else** $I \leftarrow I+1$;
assert $(\exists e' \in [I:N],\ B[e']=0)$

The verification conditions constructed by the inductive assertion method are

$$(\exists e \in [I:N],\ B[e]=0) \land B[I]=0 \rightarrow (\exists e' \in [I:I],\ B[e']=0)$$
$$(\exists e \in [I:N],\ B[e]=0) \land B[I]\neq 0 \rightarrow (\exists e' \in [I+1:N],\ B[e']=0)$$

The first may be established by choosing e' to be I. The second may be established as follows: Let e be given, so that B[e]=0 is an assumption. If e=I, then the assumption B[I]≠0 leads to a contradiction in the hypothesis of the verification condition, so the verification condition is true. Otherwise, e∈[I+1:N] so we may choose e' to be e.

We wish to formalize this by a *justification*. If B[I]=0, we have *e'←I*. Next, suppose B[I]≠0. For the case e∈[I+1:N], we write *e'*←e. The case e=I requires no instantiation of e': it suffices to construct the verification condition with e replaced by I and its truth follows without further reasoning about quantifiers. We adopt the convention that cases which can be proved without instantiation require no justification. Thus, the program with its justifications is

assert (∃e∈[I:N], B[e]=0);
if B[I]=0 **then** {N←I; **use** e'←I} **else** {I←I+1; **use if** e∈[I:N] **then** e'←e}
assert (∃e'∈[I:N], B[e']=0)

In proving an existentially quantified final assertion from an existentially quantified initial assertion, the instantiation made in the justification behaves somewhat like a normal assignment. The following interpretation can be given: Imagine control passing through the initial assertion where an oracle selects the value of the existentially quantified variable e; next, the justification expression is executed which computes a value for the existentially quantified variable e' of the final assertion; control then reaches the final assertion which must be true using the computed value for e'. This sort of description is not natural when proving a universally quantified final assertion from a universally quantified initial assertion. The choice of values for universally quantified variables goes counter to the execution flow.

Position in the formula determines whether a quantified variable is fixed or subject to instantiation and, in the case of instantiated variables, on which fixed variables they may depend. Instantiating variables as functions of other variables can be viewed as supplying the *Skolem functions* of mathematical logic. A general statement of the positional rules may be found in any standard work on logic. The following scheme summarizes the cases discussed thus far.

$$(\forall p)P(p) \land (\exists q)Q(q) \rightarrow (\forall r)R(r) \land (\exists s)S(s)$$

The variables q and r are to be fixed; p and s are to be chosen as functions of q and r.

It is convenient to use the construction **if** P **then** Q **else** R in logical formulas as an abbreviation for $(P \rightarrow Q) \wedge (\sim P \rightarrow R)$. When this construction appears in assertions and P is quantified, use of the quantified variables in justifications is described by special rules. These rules are illustrated by the following example

assert $(\forall i \in [1:I-1],\ C[i] \neq Key) \wedge L=0$;
if $C[I]=Key$ **then** $L \leftarrow I$;
assert **if** $(\forall j \in [1:I],\ C[j] \neq Key)$ **then** $L=0$ **else** $L=I$;

To understand how j is used in justifications, consider the verification conditions. The case C[I]=Key, is

$(\forall i \in [1:I-1],\ C[i] \neq Key) \wedge L=0 \wedge C[I]=Key \quad \rightarrow$
if $(\forall j \in [1:I],\ C[j] \neq Key)$ **then** $I=0$ **else** $I=I$

which may be established by proving

$$(\forall i \in [1:I-1],\ C[i] \neq Key) \wedge L=0 \wedge C[I]=Key \quad \rightarrow \quad (\exists j \in [1:I],\ C[j]=Key)$$

The variable j *is to be instantiated*; a correct justification is $j \leftarrow I$. The verification condition for the case $C[I] \neq Key$ simplifies to

$$(\forall i \in [1:I-1],\ C[i] \neq Key) \wedge L=0 \wedge C[I] \neq Key \quad \rightarrow \quad (\forall j \in [1:I],\ C[j] \neq Key)$$

The variable j *is to be fixed* and i is to be instantiated as a function of j; a correct justification is **if** $j \in [1:I-1]$ **then** $i \leftarrow j$. The program with justifications is

assert $(\forall i \in [1:I-1],\ C[i] \neq Key) \wedge L=0$;
if $C[I]=Key$ **then** {$L \leftarrow I$; **use** $j \leftarrow I$} **else** **use** **if** $j \in [1:I-1]$ **then** $i \leftarrow j$;
assert **if** $(\forall j \in [1:I],\ C[j] \neq Key)$ **then** $L=0$ **else** $L=I$;

The quantified variable j is fixed in one justification and is instantiated in the other. In each case, the justification produces a quantifier-free formula.

The situation may be summarized: Suppose a final assertion contains a conjunct **if** $(\forall j)P(j)$ **then** Q **else** R. A justification, $j \leftarrow E$, which instantiates j dictates that the proof proceed by showing $\sim P(E) \wedge R$. A justification, $i \leftarrow F(j)$, which takes j as fixed dictates that the proof proceed by showing that $P(j) \wedge Q$.

In all cases, the treatment of justifications preserves a critical property: There is an algorithm which maps a *path* (i.e. initial assertion, instructions, and final assertion) annotated with justifications into a quantifier-free formula F such that if F is true then the path is correct.

4. FORMULA LABELS

One goal of proof justifications is to allow writing programs whose proofs are clear. To aid readability, we introduce some additional, optional notation:

(1) A formula in an assertion can be labeled. For example,

assert L≤U ∧ SrtA:(∀i,j∈[L:U], i≤j → A[i]≤A[j]);

The *formula label* SrtA is a name for the second conjunct of this assertion. A well-chosen name may help convey the intent of a formula. Formula labels obey the normal block-structured scope rules and, subject to these rules, a *formula label* is unique within a scope.

(2) A justification which instantiates quantified variables can prefix the instantiation by the label of the formula in which the variables are bound. For example, the justification i←k, j←M can be written as SrtA(i←k, j←M). Since *formula labels* are unique in a scope, such a form identifies the formula being used. The similarity to a function call is intentional.

(3) A justification which establishes a formula labeled with <*formula label*> can be prefixed with the form **prove** <*formula label*> **by**, and the keyword **use** can then be omitted. Using this device to identify the formula proved by each justification may be helpful when the final assertion contains several formulas.

The following path from a binary search shows how this notation is used

```
assert L≤U ∧
    SrtA:(∀i,j∈[L:U], i≤j → A[i]≤A[j]) ∧
    KeyInLtoU:(∃k∈[L:U], A[k]=Key);
M←(L+U)/2;
if A[M]<Key then
    begin L←M+1;
    prove KeyInNewLtoU by k'←k, SrtA(i←k, j←M):k≤M→A[k]≤A[M];
    end
```

```
else
    begin U←M;
    prove KeyInNewLtoU by if k≤M then k'←k
                                   else k'←M, SrtA(i←M, j←k):M≤k→A[M]≤A[k];
    end;
prove StillSrtA by SrtA(i←i', j←j');
assert L≤U ∧
    StillSrtA:(∀i',j'∈[L:U], i'≤j' → A[i']≤A[j']) ∧
    KeyInNewLtoU:(∃k'∈[L:U], A[k']=Key);
```

The second and third formulas in the initial assertion are labeled - SrtA and KeyInLtoU. Also, two formulas in the final assertion are labeled - StillSrtA and KeyInNewLtoU. There are three justifications, each beginning with **prove**. The first, **prove** KeyInNewLtoU **by** k'←k, SrtA(i←k, j←M), may be read as follows: to prove KeyInNewLtoU, choose k' to be k; choose i and j of SrtA to be k and M. Observe how A being sorted is used to show that k' so chosen lies between the new L and U. The second justfication is similar: If k≤M, then prove KeyInNewLtoU by choosing k' to be k; if M<k then prove KeyInNewLtoU by choosing k' to be M and instantiating SrtA with i←M and j←k. The third justification proves StillSrtA by instantiating SrtA.

Formula labels are optional and redundant. So far as a mechanical verifier is concerned, **prove** KeyInNewLtoU **by** k'←k, SrtA(i←k, j←M) is a redundant statement of **use** k'←k, i←k, j←M. The programmer and subsequent readers may prefer the longer form.

In addition to its use of formula labels, this example merits attention for it illustrates three commonly occuring properties of program proofs:

(1) Proof cases refine the cases tested by executable instructions. The refinements may not be mainfest from the instructions and assertions.

(2) Instantiations may be neither manifest nor obvious.

(3) Finding the correct cases and instantiations from the instructions and assertions may require a search by either a reader or a mechanical verifier. Justifications make this search unnecesssary.

5. JUSTIFYING LOOP INVARIANTS

Justifying the correctness of a loop invariant is a special case of showing that an assertion is a logical consequence of a preceding assertion and the intervening executable instructions. Since the assertions at the beginning and end of a loop path are the same, the argument often has a stereotyped structure and *it is then possible to default part of the justification.*

For simplicity, we treat one-entrance-one-exit **while** loops. These are written as

<*while statement*> ::= **maintain** <*assertion*> **while** <*Boolean test*> **do** <*statement*>

By convention, the <*assertion*> is to hold the first time that control enters the loop, and just before the <*Boolean test*> is executed. In particular, the <*assertion*> is to hold on loops that are executed zero times. A justification of the <*assertion*> must establish: (i) that it follows from the preceding invariants, and (ii) that it remains true as control flows around the loop. As (i) has been addressed by the preceding section, we deal with (ii).

The following example takes two sorted arrays A[1:N] and D[1:M], and yields a third array C such that C[i]=A[i] if A[i] does not appear in D and C[i]=-1 otherwise. Because the arrays are sorted, this can be done in linear time with simple sequential file processing techniques.

```
I←1; J←1:
maintain SrtA:(∀p,q∈[1:N], p≤q→A[p]≤A[q]) ∧
    SrtD:(∀r,s∈[1:M], r≤s→D[r]≤D[s]) ∧
    CRule:(∀i∈[1:I-1], if DNA:(∀m∈[1:M], D[m]≠A[i]) then C[i]=A[i] else C[i]=-1) ∧
    SmallDs:(∀j∈[1:J-1], D[j]<A[I]) ∧
    I∈[1:N+1] ∧ J∈[1:M] ∧ A[N]<D[M]
while I≤N
do  if A[I]<D[J] then
        begin C[I]←A[I]; I←I+1 end
    elseif A[I]=D[J] then
        begin C[I]← -1; I←I+1 end
    elseif J<M then J←J+1
```

The executable instructions are simple; the invariant on which the correctness rests is longer. SrtA and SrtD state that A and D are sorted. CRule is the output specification with the array size N replaced by its inductive counterpart I-1. SmallDs states that each element in the sub-array D[1:J-1] is less than the element A[I] which is next to be processed. The next two conjuncts give the ranges of I and J. The final conjunct insures termination.

An informal demonstration that these formulas are invariant goes as follows: SrtA and SrtD remain true because A and D are never changed. To establish CRule and SmallDs, consider the cases tested by the code:
(1) If A[I]<D[J] then CRule requires we show that A[I] does not appear in D. SmallDs insures that A[I] does not appear in D[1:J-1]; the test A[I]<D[J] together with D being sorted, insures that A[I] does not appear in D[J:M]. SmallDs requires we show that each

element of $D[1:J-1]$ is smaller than $A[I']$ where I' is the new value of I. This follows because each element is smaller than $A[I]$, I' is I+1, and A is sorted so that $A[I] \leq A[I+1]$.
(2) If $A[I]=D[J]$ then $A[I]$ appears in D, so setting $C[I]$ to -1 is correct.
(3) If neither (1) nor (2) applies, then $A[I]>D[J]$. If $J<M$ then J can be incremented leaving SmallDs invariant.

To formalize this demonstration, we first examine the verification conditions that would result from the inductive assertion method. Consider, for simplicity, showing only that SmallDs:$(\forall j \in [1:J-1],\ D[j]<A[I])$ remains true in the case $A[I]>D[J]$.

$$\begin{array}{l}
(\forall p,q \in [1:N],\ p \leq q \rightarrow A[p] \leq A[q]) \wedge (\forall r,s \in [1:M],\ r \leq s \rightarrow D[r] \leq D[s]) \wedge \\
(\forall i \in [1:I-1],\ \textbf{if } (\forall m \in [1:M],\ D[m] \neq A[i]) \textbf{ then } C[i]=A[i] \textbf{ else } C[i]=-1) \wedge \\
(\forall j \in [1:J-1],\ D[j]<A[I]) \wedge I \in [1:N+1] \wedge J \in [1:M] \wedge A[I] \geq D[J] \wedge A[N]<D[M] \wedge \\
A[I] \neq D[J] \quad \rightarrow \\
(\forall j \in [1:J],\ D[j]<A[I])
\end{array}$$

To prove this, let j in the conclusion be fixed. If that j lies in the interval $[1:J-1]$, choose j of the assumption to be that j. If that j is J, then the assumptions $A[I] \geq D[J]$ and $A[I] \neq D[J]$ imply $A[I]>D[j]$.

In this description, it is vital to distinguish the j of the conclusion from the j of the assumption. For formulas of this class, we write $j\downarrow$ to mean the j of the conclusion and use j without suffix to mean the j of the assumption. The expression $j\downarrow$ may be read as "j at the path end". Further, we can use the convention, introduced in the preceding section, that if a formula can be proved without quantification then no explicit justification need be written, so that the proof of $D[J]<A[I]$ does not need to be justified. Thus, the formal justification is: **if** $j\downarrow \in [1:J-1]$ **then** $j \leftarrow j\downarrow$.

We are now prepared to introduce *default* justifications. Universally quantified loop invariants often have justifications similar to the one above, i.e. for the sub-range unaffected by the ith loop iteration, one proves $P(j\downarrow)$ by using $P(j)$. We adopt this as the default: *Whenever a sub-range of a universally quantified loop invariant P(j) occurs on the i-th and i+1st iteration and the quantified variable j is not explicitly instantiated by a justification, then the choice $j \leftarrow j\downarrow$ will be made.*

Using this default rule and the convention that proofs which proceed without quantification need no justification, the above verification condition requires no explicit justification. This situation is not uncommon. For example, the justification for SrtA:$(\forall p,q \in [1:N],\ p \leq q \rightarrow A[p] \leq A[q]) \wedge$ SrtD:$(\forall r,s \in [1:N],\ r \leq s \rightarrow D[r] \leq D[s])$, is $p \leftarrow p\downarrow$; $q \leftarrow q\downarrow$; $r \leftarrow r\downarrow$; $s \leftarrow s\downarrow$. As this is the default instantiation, it can be omitted.

We next treat a proof which partially defaults. Consider showing that SmallDs:(∀j↓∈[1:J-1], D[j↓]<A[I↓]) in the case A[I]<D[J], so the value of I at the path end, I↓, is I+1. By assumption, A is sorted, i.e. SrtA:(∀p,q∈[1:N], p≤q→A[p]≤A[q]). Choose p to be I↓-1 and q to be I↓, giving the assumption A[I↓-1]≤A[I↓], i.e. A[I]≤A[I↓]. The default instantiation j←j↓, establishes D[j↓]<A[I]. Thus, D[j↓]<A[I↓].

Consider showing that CRule:(∀i∈[1:I-1], **if** DNA:(∀m∈[1:M], D[m]≠A[i]) **then** C[i]=A[i] **else** C[i]=-1) is an invariant in the case A[I]<D[J]. Let I↓ denote the value of I at the path end. Suppose i↓∈[1:I↓-2]. This is covered by choosing i←i↓, which is the default. The remaining case is i↓=I↓-1. In section 2, we showed that (∀m↓∈[1:M], D[m↓]≠A[I↓-1]) using a justification which may be written

```
if i↓=I↓-1 then
    if m↓∈[1:J-1] then SmallDs(j←m↓): D[m↓]<A[I↓-1]
    elseif m↓∈[J:M] then SrtD(r←J, s←m↓): D[J]≤D[m↓]
```

We adopt the convention that an ordinary variable such as I appearing in a justification is interpreted as denoting the value of that variable at the point where the justification is written. If the above justification is placed in the program after I is incremented, then I has its final value and *I* can be used in place of *I*↓ with no change in meaning.

Only one other case requires an explicit justification. Suppose A[I]=D[J]. Since C[I] is set to -1, CRule requires that we prove (∃m↓∈[1:M], D[m↓]=A[I]). The justification is m↓←J, which may be written DNA(m↓←J).

The program with its justifications is

```
I←1; J←1:
maintain SrtA:(∀p,q∈[1:N], p≤q→A[p]≤A[q]) ∧
    SrtD:(∀r,s∈[1:M], r≤s→D[r]≤D[s]) ∧
    CRule:(∀i∈[1:I-1], if DNA:(∀m∈[1:M], D[m]≠A[i]) then C[i]=A[i] else C[i]=-1) ∧
    SmallDs:(∀j∈[1:J-1], D[j]<A[I]) ∧
    I∈[1:N+1] ∧ J∈[1:M] ∧ A[N]<D[M]
while I≤N
do  if A[I]<D[J] then
        begin C[I]←A[I]; I←I+1;
        prove SmallDs by SrtA(p←I-1, q←I): A[I-1]≤A[I];
        prove CRule by if i↓=I-1 then
            if m↓∈[1:J-1] then SmallDs(j←m↓): D[m↓]<A[I-1]
            elseif m↓∈[J:M] then SrtD(r←J, s←m↓): D[J]≤D[m↓]
        end
```

```
elseif A[I]=D[J] then
    begin C[I]← -1; I←I+1;
    prove CRule by if i↓=I-1 then DNA(m↓←J)
    end
elseif J<M then J←J+1
```

The overall proof structure may be seen by inspection: SmallDs is proved using SrtA; CRule is proved using SmallDs and SrtD.

To conclude the discussion of loops, we observe that **for** statements constitute a particularly regular class. The typical **for** statement has the form

maintain ∀i∈[L:I-1], P(i) **for** I **from** L **to** U **do** S(I)

The invariant specifies that P(i) has been achieved for each i in the subrange [L:I-1] that has been processed. Each iteration establishes P(I) and leaves undisturbed P(i) for i∈[L:I-1]. *The default justifications match this typical* **for** *loop exactly and justifications are unnecessary.* Where **for** loops do not have this form and justifications are needed, their appearance is useful documentation - indicating an irregular situation.

6. DEFINITIONS AND LEMMAS

Programs and their proofs may be sharpened by introducing definitions to articulate important abstractions. As an example, we discuss part of the fast string searching algorithm of [6]. It takes a pattern P[1:M] and a text string T[1:N] and finds the first place in T where P matches - in linear time. To express the correctness specification, it is convenient to define the predicate *Match.* We define new predicates with the form

predicate *<predicate name>*(*<formal parameters>*): *<assertion>*;

We define what it means for *the substring P[1:m] to Match T ending at position s*

predicate Match(P,m,T,s; j): $m \leq s \wedge (\forall j \in [1:m-1], P[j]=T[s-m+j])$;

Parameters following the semicolon are optional. If present, they give local names to bound variables for use in justifications, as explained below.

The algorithm uses two integer variables J and K to index P and T respectively. These indices are initialized to 1 so initially, Match(P,J,T,K) is vacuously true. Throughout the search, the invariant Match(P,J,T,K) is maintained. If J ever reaches M+1, a complete

match for P in T has been found. (A complete specification would additionally require that the match so found is the leftmost complete match. In the interest of brevity, we defer this part of the specification and its justification to the appendix.)

The linear search time is attained by first processing the pattern P to construct a table L of longest initial matches. For each $m \in [1:M]$, L[m] is the longest initial substring of P which matches a proper tail of P[1:m]. More precisely, L is constructed so that

```
LSpec:(∀m∈[1:M],
    L[m]∈[0:m-1] ∧ Match(P,L[m],P,m) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,i,P,m)))
```

Since L and P are never changed, this remains invariant.

At each step of the search, there are two cases. If P[J]=T[K], then J and K are each incremented by 1, maintaining the invariant Match(P,J,T,K). The program fragment is

```
assert Match(P,J,T,K;j1) ∧
    LSpec:(∀m∈[1:M],
        L[m]∈[0:m-1] ∧ Match(P,L[m],P,m) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,i,P,m)));
if P[J]=T[K] then {J←J+1; K←K+1};
assert Match(P,J,T,K;j2) using if j2∈[1:J-2] then j1←j2;
```

This shows how the optional parameters to Match are used: The final assertion is Match(P,J,T,K). As Match contains a universally quantified formula of the form $(\forall j \in [1:J-1], Q(j))$, we show how to prove Q(j) for each $j \in [1:J-1]$ - using a justification. The justification must refer to the quantified variable j in two occurrences of Match. We give j two distinct local names by supplying the optional parameters in these two occurrences.

The other case is $P[J] \neq T[K]$. J is set to L[J], so the new value of J indexes the longest preceding partial match. This also maintains the invariant Match(P,J,T,K). The program fragment with justifications is

```
assert Match(P,J,T,K;j1) ∧
    LSpec:(∀m∈[1:M],
        L[m]∈[0:m-1] ∧ Match(P,L[m],P,m;j2) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,m,P,i)));
if P[J]≠T[K] then
    begin use j1←J-L[J]+j3: J≤K ∧ P[J-L[J]+j3]=T[K-L[J]+j3];
    use m←J, j2←j3: L[J]≤J ∧ P[j3]=P[J-L[J]+j3];
    J←L[J]
    end
```

assert Match(P,J,T,K;j3): J≤K ∧ (∀j3∈[1:J-1], P[j3]=T[K-J+j3])

As before, optional parameters to Match in the assertions are used in the justification. The formulas following the colons show the results of the instantiations.

Although the proof is correct, it can be improved. Because it appeals too directly to definitions, it obscures a simple argument: Match is a transitive relation, so if P[1:L[J]] matches P at J and P [1:J] matches T at K then P[1:L[J]] matches T at K. A better proof would first establish, as a lemma, that Match is a transitive relation and then use this property in the main proof.

To introduce a lemma, we write

lemma ⟨*lemma name*⟩(⟨*formal parameters*⟩): ⟨*assertion*⟩;

As elsewhere, the ⟨*assertion*⟩ may include a justification. The ⟨*assertion*⟩ is a formula *to be proved*, using the justification if one is present. The transitivity result for Match may be stated and proved:

lemma TransMatch(A,x,B,y,C,z):
Match(A,x,B,y;j1) ∧ Match(B,y,C,z;j2) → Match(A,x,C,z;j3)
using j1←j3, j2←y-x+j3;

Using a separate lemma has two benefits: (i) a general result is obtained; (ii) only relevant assumptions need be considered when checking its truth. Since optional parameters to predicates serve only to aid the justifications, a lemma which is shown valid with optional parameters is valid if the optional parameters are deleted. Thus, this lemma establishes (∀A,x,B,y,C,z) Match(A,x,B,y) ∧ Match(B,y,C,z) → Match(A,x,C,z).

A valid formula such as this may be added to the assumption of a verification condition with any choice for the universally quantified variables A,x,B,y,C,z. It is convenient to invoke lemmas by name and describe the instantiations by actual parameters to the lemma invocation. For example, the above fragment may be rewritten

assert Match(P,J,T,K) ∧
    LSpec:(∀m∈[1:M],
        L[m]∈[0:m-1] ∧ Match(P,L[m],P,m) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,m,P,i)));
if P[J]≠T[K] **then**
    begin use LSpec(m←J): Match(P,L[J],P,J);
    use TransMatch(P,L[J],P,J,T,K):
        Match(P,L[J],P,J) ∧ Match(P,J,T,K) → Match(P,L[J],T,K);

```
    J←L[J]
    end;
assert Match(P,J,T,K)
```

Here, the role of transitivity is clear. Using a lemma in this way articulates an important property of Match and sharpens the argument.

7. OTHER FUNCTIONALS

We have shown how justifications can be used to obtain quantifier-free verification conditions from assertions with quantified formulas. In the context of program verification, it is enlightening to regard the quantifiers $\forall$ and $\exists$ as merely two members of a collection of built-in functionals which may appear in assertions. Other functionals in the collection are: *theleast*, *sum*, *product*, and *cardinality*. The quantifiers and these other functionals are similar in that they take a formula or term as a parameter and apply that parameter to some set of values.

Like the quantifiers, these other functionals can make theorem proving difficult. As with the quantifiers, we use *justifications* to map verification conditions with functionals into *simpler* theorems whose truth implies the original verification conditions. The form of justifications depends on the functional. The meaning of *simpler* also depends on the functional: for the quantifiers, *simpler* means quantifier-free; for other functionals, *simpler* usually means that the new theorems can be proved by treating any remaining occurrences of functionals as atomic terms. This is best explained with an example.

The functional *cardinality* is representative. We write $\mathbb{C}(j\in[i:f], P(j))$ to denote the cardinality of the set $\{j \mid j\in[i:f] \wedge P(j)\}$. The following may be said of $\mathbb{C}$:

(R1) $\mathbb{C}(j\in[i:f], P(j)) = 0$, if $i>f$

(R2) $\mathbb{C}(j\in[i:f], P(j)) =$ (**if** $P(j)$ **then** 1 **else** 0), if $i=f$

(R3) $\mathbb{C}(j\in[i:f], P(j)) =$
$\mathbb{C}(j\in[i:r-1], P(j)) + \mathbb{C}(j\in[r:s], P(j)) + \mathbb{C}(j\in[s+1:f], P(j))$, if $i\leq r\leq s\leq f$

(R4) $\mathbb{C}(j\in[i:f], P(j)) \leq \mathbb{C}(j\in[i:f], Q(j))$, if $(\forall j\in[i:f], P(j)\rightarrow Q(j))$

A verifier in which $\mathbb{C}$ is built-in and which suitably treats equality and inequality can use R1 and R2 as reduction rules, invoking them automatically. It is difficult, however, to invoke R3 automatically. Like the universally quantified formula $(\forall s\in[i:f], P(s))$, R3 can be instantiated for any s in the range. Knowing when to invoke R3 and choosing the right values for s and r is not a mechanical process. Justifications seem appropriate.

One may regard R3 (and R4) as built-in lemmas whose truth is assumed.

lemma SplitC(j,i,f,r,s,P):

$$i \leq r \leq s \leq f \quad \rightarrow$$
$$\mathbb{C}(j \in [i:f], P(j)) = \mathbb{C}(j \in [i:r-1], P(j)) + \mathbb{C}(j \in [r:s], P(j)) + \mathbb{C}(j \in [s+1:f], P(j));$$

lemma OrderedC(j,i,f,P,Q):

$$(\forall j \in [i:f], P(j) \rightarrow Q(j)) \rightarrow \mathbb{C}(j \in [i:f], P(j)) \leq \mathbb{C}(j \in [i:f], Q(j));$$

Thus, these rules may be invoked with the same syntax and with the same interpretation as lemmas in the preceding section. For example,

using SplitC(k,1,N,L-1,L,A[k]≤0)

use OrderedC(k,2,M,A[k]=0,B[k]<k):

$$(\forall k \in [2:M], A[k]=0 \rightarrow B[k]<k) \rightarrow \mathbb{C}(k \in [2:M], A[k]=0) \leq \mathbb{C}(k \in [2:M], B[k]<k)$$

Note that some formal parameters are predicate letters, i.e. the lemmas are second-order formulas. Since they will be used only when explicitly invoked with actual parameters, their instances will be first-order formulas.

Invoking these rules instantiates the rule body and adds it to the verification condition. As always, the occurence of an ordinary variable in a justification refers to the current value of that variable. For example,

```
assert L∈[1:N] ∧ ℂ(j∈[1:N], B[j]=0) ≥ K;
if B[L]>2 then
    begin use SplitC(j,1,N,L,L,B[j]=0);
    B[L]←B[M]-B[N];
    use SplitC(n,1,N,L,L,B[n]=0);
    end;
assert ℂ(n∈[1:N], B[n]=0) ≥ K;
```

The invocation SplitC(j,1,N,L,L,B[j]=0) refers to the initial value of B. The invocation after the assignment, i.e. SplitC(n,1,N,L,L,B[n]=0), refers to the final value of B. Using these justifications, the verification condition simplifies to

$$\mathbb{C}(j \in [1:L-1], B[j]=0) + (\textbf{if } B[L]=0 \textbf{ then } 1 \textbf{ else } 0) + \mathbb{C}(j \in [L+1:N], B[j]=0) \geq K \wedge B[L]>2 \rightarrow$$
$$\mathbb{C}(j \in [1:L-1], B[j]=0) + (\textbf{if } B[M]=B[N] \textbf{ then } 1 \textbf{ else } 0) + \mathbb{C}(j \in [L+1:N], B[j]=0) \geq K$$

This may be proved by treating $\mathbb{C}(j \in [1:L-1], B[j]=0)$ and $\mathbb{C}(j \in [L+1:N], B[j]=0)$ as atomic

terms, i.e. as new variables x and y, obtaining the formula

x+(if B[L]=0 **then** 1 **else** 0)+y $\geq$ K $\wedge$ B[L]>2 $\rightarrow$
x+(if B[M]=B[N] **then** 1 **else** 0)+y $\geq$ K

Further concern with the meaning of $\mathbb{C}$ is unnecessary.

8. PROOFS WHICH NEED NO JUSTIFICATION

In general, proof justifications are necessary. There are, however, certain classes of theorems for which a suitably fast decision procedure is known. For such classes, justifications are not necessary. It is important to understand these classes, in order to put justifications into proper perspective. Universally quantified Presburger arithmetic is one such classs: there is no need to annotate 3*I<B[J] $\wedge$ B[J]<K+1 $\rightarrow$ 3*I<K+1 with a justification using the transitivity of less-than. However, other well-defined classes having a decision procedure are less widely appreciated. *Such classes deserve attention and should be exploited.*

A fundamental result is found in [8]. It may be stated informally: *If F is a recursively defined total function, V is a formula which contains a single appearance of F, and V specifies F uniquely, then subgoal induction can be used to construct another formula V' free of F such that* $V \equiv V'$. This provides a way of eliminating function letters from verification conditions. Using it, we may obtain a formula for which a decision procedure is known. Any formula which, by inspection, may be seen to have this property needs no justification.

As an example, we treat the functional Sum. For the purposes of an induction proof, it may be defined by the recursive procedure

procedure Sum(i,n,E): **if** i>n **then** 0 **else** E(n)+Sum(i,n-1,E);

Consider the theorem
$n \geq 0 \rightarrow \mathrm{Sum}(1, n, (\lambda j)j^3) = n^4/4 + n^3/2 + n^2/4$

This might appear difficult to establish automatically. However, inspection of its form reveals that the above result applies. One may apply the rule of subgoal induction to the definition of Sum and mechanically construct the equivalent theorem

$(n \geq 0 \wedge 1>n \rightarrow 0 = n^4/4+n^3/2+n^2/4) \wedge$
$(n \geq 0 \wedge 1 \leq n \wedge z= (n-1)^4/4+(n-1)^3/2+(n-1)^2/4 \rightarrow n^3+z = n^4/4+n^3/2+n^2/4)$

in confidence that proof of the new theorem is merely a matter of tedious algebraic manipulation. The form of the original theorem insures that no creativity will be required.

9. CONCLUSION

It has been our aim to show that proof justifications should be an integral part of their programs. There are three reasons for this. The first reason comes from mathematical logic: There is no decision procedure for first order predicate calculus. Without justifications, a mechanical program verifier cannot guarantee that in finite time it will verify each correct program and reject each incorrect one.

The second reason comes from software engineering practice. Programs must be read and understood by programmers who did not write the code, e.g. co-workers who later modify it for changed needs. When quantified assertions must be proved by making the right instantiations, it may be difficult for a reader to understand the chain of reasoning which establishes a verification condition. Justifications are good documentation for anyone who wishes to understand why a program works.

The third reason comes from programming methodology. The preceding considerations regarding other programmers apply in part to the original programmer. When assertions involve quantification, it may be difficult for him to be confident that the assertions he supplies are complete. If they are not, mechanical verification may fail even though the program is correct. As long as the proof process is not articulated, constructing the right assertions will be regarded as mysterious and error-prone. By providing a notation in which proofs can be justified with precision and rigor, we hope to replace this attitude with the understanding that verification can be made systematic and lucid.

10. ACKNOWLEDGMENTS

I am indebted to Dan Bobrow, Peter Deutsch, Jim Morris, Lyle Ramshaw and Nori Suzuki for constructive criticism of these ideas and their exposition in this paper.

REFERENCES

1. Deutsch, L. P. *An Interactive Program Verifier*, Ph. D. Thesis, U. of Calif., Berkeley, Ca., 1973.

2. Floyd, R. W. Assigning Meanings to Programs, in *Mathematical Aspects of Computer Science*, J. T. Schwartz (Ed.), AMS, (1967), pp. 19-32.

3. Good, D. I. London, R. L. and Bledsoe, W. W. An interactive program verification system, *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975) pp. 59-67.

4. Hoare, C. A. R. Procedures and Parameters: An Axiomatic Approach, in *Symposium on Semantics of Algorithmic Languages*, E. Engler (Ed.), Springer-Verlag, 1970, pp. 102-116.

5. King, J. C. *A Program Verifier*, Ph. D. Thesis, Carnegie-Mellon U., Pittsburgh, Pa., 1969.

6. Knuth, D. E., Morris, J. H., and Pratt, V. R. *Fast Pattern Matching in Strings*, CS 440, Computer Science Dept., Stanford U., Aug. 1974.

7. Manna, Z. The Correctness of Programs, *JCSS*, 3 (1969), pp. 119-127.

8. Morris, J. H. and Wegbreit, B. Subgoal Induction, to appear in *CACM.*

9. Suzuki, N. Verifying Programs by Algebraic and Logical Reduction, *Int. Conf. on Reliable Software*, Los Angeles, Ca. (April 1975) pp. 473-481.

10. Topor, R. W. *Interactive Program Verification Using Virtual Programs*, Ph. D. Thesis, University of Edinburgh, 1975.

11. Wegbreit, B. The Synthesis of Loop Predicates, *CACM*, Vol. 17, No. 2 (Feb. 1974), pp. 102-112.

APPENDIX: AN EXTENDED EXAMPLE

The utility of proof justifications can best be appreciated by studying in depth a program which requires a subtle proof. We finish verifying the linear string searching algorithm [6] started in section 6. The algorithm is to find the leftmost position in the text T[1:N] which matches the pattern P[1:M]. Section 6 describes the method in some detail; a rereading of that section may be helpful. We defined

predicate Match(P,m,T,s; j): $m \leq s \wedge (\forall j \in [1{:}m-1],\ P[j]=T[s-m+j])$;

We assume the pattern P has been processed to constructe a table L so that for each m∈[1:M], L[m] is the longest initial substring of P which matches a proper tail of P[1:M], i.e.

```
LSpec:(∀m∈[1:M],
        L[m]∈[0:m-1] ∧ Match(P,L[m],P,m) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,i,P,m)));
```

To write the specifications compactly, we declare that LSpec is a global **invariant,** meaning that it holds *throughout* the scope and is implicitly added to each assertion. With this notation, the program is

```
invariant LSpec:(∀m∈[1:M],
            L[m]∈[0:m-1] ∧ Match(P,L[m],P,m) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,i,P,m)));
J←1; K←1;
maintain Match(P,J,T,K) ∧ NotSeen1:(∀s∈[M+1:M+K-J], ~Match(P,M+1,T,s))
while J≤M ∧ K≤N
do  begin

    maintain Match(P,J,T,K) ∧ NotSeen2:(∀t∈[M+1:M+K-J], ~Match(P,M+1,T,t))
    while J>0 ∧ P[J]≠T[K]
    do J←L[J];

    K←K+1; J←J+1
    end
```

The outer loop advances J and K together, scanning forward in the pattern P and text T. The inner loop backs up J, via J←L[J], whenever corresponding characters in P and T do not match.

The correctness specification given in section 6 is augmented here by the requirement that (∀t∈[M+1:M+K-J], ~Match(P,M+1,T,t)) i.e., no complete match has been seen. The hardest part of the verification is showing that this is maintained in the case $J>0 \wedge P[J]\neq T[K]$. Because the assignment J←L[J] extends the interval [M+1:M+K-J], showing that the invariant is maintained requires a non-trivial argument. The reader may find it instructive to carry out the verification himself before reading further.

The justification we use depends on two lemmas. The first lemma states that if a pattern matches a text at some position then any initial subpattern matches the text at the corresponding position, i.e.

```
lemma SubMatch(P,U,T,K,S):
Match(P,U,T,K; j1) ∧ S≤U → Match(P,S,T,K-U+S; j2)
using j1←j2;
```

The second lemma states that if two patterns match a text at a common place, then the two patterns match each other, i.e.

```
lemma Common(P,J,Q,X,T,K):
Match(P,J,T,K; j1) ∧ Match(Q,X,T,K; j2) ∧ X≤J → Match(Q,X,P,J; j3)
using j1←J-X+j3, j2←j3;
```

The program fragment which uses the lemma is

```
assert Match(P,J,T,K) ∧
    (∀t∈[M+1:M+K-J], ~Match(P,M+1,T,t; j)) ∧
    LSpec:(∀m∈[1:M],
            L[m]∈[0:m-1] ∧ Match(P,L[m],P,m) ∧ (∀i∈[L[m]+1:m-1], ~Match(P,i,P,m)));
if J>0 ∧ P[J]≠T[K] then
    begin
    use if t'∈[M+1:M+K-J] then {t←t'; j'←j}
        elseif t'=M+K-J+1 then j'←J
        elseif t'∈[M+K-J+2: M+K-L[J]] then
            SubMatch(P,M+1,T,t',M+1-t'+K):
                Match(P,M+1,T,t') ∧ M+1-t'+K≤M+1 → Match(P,M+1-t'+K,T,K),
            Common(P,J,P,M+1-t'+K,K):
                Match(P,J,T,K) ∧ Match(P,M+1-t'+K,T,K) ∧ M+1-t'+K≤J →
                Match(P,M+1-t'+K,P,J),
            LSpec(m←J, i←M+1-t'+K):
                ~Match(P,M+1-t'+K,P,J);
    J←L[J];
    end
assert (∀t'∈[M+1:M+K-J], ~Match(P,M+1,T,t'; j'))
```

This may be read as: To show that (∀t'∈[M+1:M+K-J], ~Match(P,M+1,T,t'; j')) consider 3 cases.

(1) If t'∈[M+1:M+K-J] use the second initial assertion and choose j' of the final assertion to be j.

(2) If t'=M+K-J+1 use the program test P[J]=T[K] by choosing j' to be J.

(3) If t'∈[M+K-J+2: M+K-L[J]] then invoke the lemmas SubMatch and Common to show that Match(P,M+1,T,t') leads to a contradiction, which proves ~Match(P,M+1,T,t').