ASSIGNMENT TO SUBSCRIPTED VARIABLES

David Gries

TR 77-305

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

Assignment to Subscripted Variables

by

David Gries
Computer Science Department
Cornell University
September 28, 1976

Abstract

The assignment b[r]:=e is investigated using two axiomatic definitions, in order to gain an understanding of the problems involved with using arrays. It is seen that assignment to array elements leads to many of the difficulties encountered with pointers or references. The axiomatic definition is extended to cover the multiple assignment statement to both simple and subscripted variables, and a proof of correctness for a nontrivial program is outlined using the new definition.

This research was supported in part by the National Science Foundation under grant NSF GJ-42512. The author is grateful to IFIP working group 2.3 (programming methodology) for the opportunity to present and discuss this material at its July 1976 meeting. Thanks also to E.W. Dijkstra, J. Donahue and J. Horning for their constructive criticisms of earlier drafts.

1. Introduction

Assignment to simple variables is well understood, and educated researchers and programmers are actually using the axiomatic definition given by Hoare [69]. Using the notation R^x_e to denote simultaneous textual replacement of all occurrences of x in assertion R by e, and using the notion of "weakest precondition" developed by Dijkstra [76], assignment x:=e is defined by[+]

(1.1) $wp("x:=e", R) = R^x_e$

Simultaneous, multiple assignment to different simple variables x1,...,xn is similarly defined by

(1.2) $wp("x1,\ldots,xn := e1,\ldots,en", R) = R^{x1,\ldots,xn}_{e1,\ldots,en}$

Use of an axiomatic definition for assignment to subscripted variabl is less widespread. Hoare and Wirth [73] give a good, simple axiomatic definition, but one rarely sees proofs of program correctness using it. By and large, programs using arrays are still understood at a vague, informal level.

The purpose of this paper is to investigate assignment to subscripted variables, in order to make both language designers and programmers aware of the problems of working with arrays. We will review (in Section 2) the Hoare-Wirth definition and give examples of its use.

[+]For clarity and brevity, we omit necessary requirements on types of variables, domains of expressions, etc. For the same reasons, we also restrict our attention to one-dimensional arrays.

Their definition is so disarmingly simple that it hides most of the problems we have with arrays. Arrays can be used to perform most of the tasks that pointers are used for -- to implement linked lists, trees, graphs, etcs. -- and careless use of arrays is just as dangerous and unreliable as careless use of pointers.

In section 3, we give a second, equivalent definition of assignment to arrays, which brings to light the difficulties involved in handling them. The two definitions are compared. Informally, we attempt to show that the complexity involved in dealing with arrays can go up exponentially with the number of different "names" (e.g. A[i], A[j]) that could possible refer to the same element of the array in the postcondition of a subscripted variable assignment. This enforces the general opinion that uncontrolled "aliasing" -- using pointers, arrays, FORTRAN EQUIVALENCE, referencing the same variable via a parameter and via a global variable, or what have you -- leads all too easily to unmanageable programs. Thus the language designer must be extremely cautious in introducing such concepts into the language.

These first few sections attempt to provide understanding to language designers of existing knowledge. The "original" part of this paper is Sections 4 and 5. Here, we present for the first time an axiomatic definition for the multiple assignment statement to subscripted variables (e.g. x,b[i], b[j]:= e1,e2,e3). The multiple assignment is typically used to avoid sequential overspecification, and we will present a correctness-proof outline for a nontrivial program to illustrate this. Interestingly, the axiomatic definition provides a deeper and better understanding of the multiple assignment and its capabilities, then has the conventional operational definition.

2. The Definition of Subscripted Variable Assignment

Given a function f, the notation (f; i:v) refers to the function defined by

$$(2.1)\quad (f;\ i{:}v)[j] = \underline{\text{if}}\ j = i\ \underline{\text{then}}\ v\ \underline{\text{else}}\ f[j]$$

In a similar manner, we extend this notation to redefining the function at several values of its domain; for example

$$(2.2)\quad (f;\ i1{:}v1;\ i2{:}v2)[j] = \begin{cases} j=i2 \rightarrow v2 \\ j\neq i2\ \underline{\text{and}}\ j=i1 \rightarrow v1 \\ j\neq i2\ \underline{\text{and}}\ j\neq i1 \rightarrow f[j] \end{cases}$$

Hence, the order of the pairs (i1,v1) and (i2,v2) is important if (and only if) i1 = i2.

We use the notation (f; i1,i2,...,im: v1,v2,...,vm) to mean the function

$$(2.3)\quad (f;\ i1,\ldots,im{:}\ v1,\ldots,vm)[j] = \begin{cases} j=i1 \rightarrow v1 \\ \cdot \\ \cdot \\ \cdot \\ j=im \rightarrow vm \\ j\neq i1\ \underline{\text{and}}\ldots\underline{\text{and}}\ j\neq im \rightarrow f[j] \end{cases}$$

Note that this is a function only if $i_k=i_\ell \Rightarrow v_k=v_\ell$, for $1\le k<\ell\le m$.

An array b[1:n] is considered to be a partial function with domain {1,...,n} and range the set of values assignable to array elements. The assignment b[r]:=e for expressions r and e is then defined as

(2.4) $wp(\text{"}b[r]:=e\text{"}, R) = R^{b}_{(b;\ r:\ e)}$

Hence, the precondition is the postcondition R, with the function b replaced everywhere by the function (b; r: e). This definition may be understood as follows. Assertion R, which contains references b[i], must be true after execution of the assignment. Any reference b[i] in R for which i=r (the value of r <u>before</u> execution) refers to the value of e (<u>before</u> execution of the assignment); all other re-references b[i] in R refer to the value of b[i] before execution of the assignment. Any reference b[i] in the precondition must have the same property. This is done by replacing references b[i] by (b; r: e)[i], or in other words by replacing the function b by the function (b, r: e).

A few simple examples should help:

$$\begin{aligned} wp(\text{"}x[j]:=j\text{"}, \{x[j]=j\}) &\equiv (x;j:j)[j]=j \\ &\equiv j=j \\ &\equiv \underline{\text{true}} \end{aligned}$$

$$\begin{aligned} & wp(\text{"}b[j]:=0\text{"}, \{b[i]=0 \text{ for } 1\le i\le n\}) \\ &\qquad \equiv (b;j:0)[i]=0 \text{ for } 1\le i\le n \\ &\qquad \equiv b[i]=0 \text{ for } 1\le i\le n,\ i\ne j \end{aligned}$$

$$\begin{aligned} & wp(\text{"}b[2]:=b[1]\text{"}, \{b[i]= b[1] \text{ for } 1\le i\le n\}) \\ &\qquad \equiv (b;2:b[1])[i]=(b;2:b[1])[1] \text{ for } 1\le i\le n \\ &\qquad \equiv (b;2:b[1])[i]=b[1] \text{ for } 1\le i\le n \\ &\qquad \equiv b[i]=b[1] \text{ for } 2<i\le n \end{aligned}$$

In simplifying the precondition in each case. We are using the definition (2.1) of a function (b;i:v). Thus, in the first case, we have

$$
\begin{aligned}
& (x;j:j)[j] = j \\
\equiv\ & (\underline{\text{if}}\ j=j\ \underline{\text{then}}\ j\ \underline{\text{else}}\ x[j]) = j \\
\equiv\ & (\underline{\text{if}}\ \underline{\text{true}}\ \underline{\text{then}}\ j\ \underline{\text{else}}\ x[j]) = j \\
\equiv\ & j=j \\
\equiv\ & \underline{\text{true}}
\end{aligned}
$$

In return for having a simple assignment statement definition, we have to perform some (at times) rather messy manipulation of the precondition using the rules of logic given for the assertion language in order to simplify the precondition.

The next three examples are not as obvious as the previous three. The reader is encouraged to attempt to determine the weakest precondition using his own informal techniques before studying our solutions; this should bring appreciation for the need for the more formal and reliable approach.

(2.5) $wp(\text{"}x[i]:=1\text{"}, \{x[i]=x[j]\})$

$$
\begin{aligned}
&\equiv (x;i:1)[i] = (x;i:1)[j] \\
&\equiv 1 = (x,i:1)[j] \\
&\equiv 1 = (\underline{\text{if}}\ i=j\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x[j]) \\
&\equiv i=j\ \underline{\text{or}}\ x[j]=1
\end{aligned}
$$

(2.6) $wp(\text{"}x[i]:=c1\text{"}, \{x[x[i]]=c2\})$ (for constants c1, c2)

$$
\begin{aligned}
&\equiv (x;i:c1)[(x;i:c1)[i]] = c2 \\
&\equiv (x;i:c1)[c1] = c2 \\
&\equiv (\underline{\text{if}}\ i=c1\ \underline{\text{then}}\ c1\ \underline{\text{else}}\ x[c1]) = c2 \\
&\equiv (i=c1\ \underline{\text{and}}\ c1=c2)\ \underline{\text{or}}\ (i\neq c1\ \underline{\text{and}}\ x[c1]=c2)
\end{aligned}
$$

(2.7) wp("x[x[j]]:=j", {x[j]=j})
  $\equiv$ (x;x[j]:j)[j] = j
  $\equiv$ (if x[j]=j then j else x[j]) = j
  $\equiv$ (x[j]=j and j=j) or (x[j]≠j and x[j]=j)
  $\equiv$ x[j]=j

3. A second viewpoint on subscripted variable assignment

By viewing an array b as a function, and viewing assignment to b[i] as a complete change of b instead of as a change in one element of b, Hoare and Wirth were able to arrive at the simple and elegant definition (2.4). This seems to be the right viewpoint, both for the language designer and the programmer. We now rework definition (2.4) into an equivalent one which views the assignment as changing just one element of the array - that is, one subscripted variable. This is how the programmer typically views array assignment.

As a first step, we use a simple trick to transform the post-condition R into a simpler form. Suppose we wish to determine the precondition wp("v1,...,vn:=e1,...en",R) where each of the vi are simple or subscripted variables. We require that

(3.1) the postcondition may contain no nested references to the simple variables or arrays being assigned to.

For example, a postcondition of an assignment x,b[i]:=e1,e2 which contains b[x+b[x]], must be transformed to elide the nested references to both x and b.

Suppose postcondition R contains a subexpression b[f(b[s])] where f is some function of b[s]. Then R can be transformed into the equivalent assertion

$$R' \equiv R_t^{f(b[s])} \ \underline{\text{and}}\ t = f(b[s]). \qquad (t \text{ not in } R)$$

Since t is not a program variable but just a name introduced for convenience sake, we henceforth <u>always</u> emphasize its nature by writing

$$R' \equiv R_t^{f(b[s])} \ \underline{\text{with}}\ t = f(b[s]).$$

Intelligent iterative application of this <u>unnesting rule</u> yields an assertion with no nested references. For example, we transform R containing b[x+b[x]] to unnest x and b into

$$R' \equiv \ldots b[t2+t1]\ldots \underline{\text{with}}\ t1=b[t2], t2=x$$

As a final example, suppose postcondition R contains a subexpression indicating that a sequence of values ℓ contains the values of a linked list implemented using an array b, and variable p:

$$\ell = (p, b[p], b[b[p]], \ldots, b^m[p])$$

In order to unnest b and p, we transform this into

$$\ell = (to, t1, \ldots, tm)\ \underline{\text{with}}\ t0=p,\ t1=b[t0], \ldots, tm=b[t_{m-1}].$$

We emphasize that this little trick may simplify the process of determining the weakest precondition simply by reducing the number of array references b[i] in the postcondition. In this linked list example, the number of array references is essentially reduced from $O(m^2)$ to m+1.

We now begin our revision of the definition of assignment b[r]:=e, under the condition that the postcondition R satisfies (3.1), and secondly under the condition that all references in R to array b have one of the n forms b[s1],...,b[sn] (each may occur many times). First, we rewrite the assignment b[r]:=e as

t:=r; b[t]:=e

where t is a fresh variable. Then, noticing that because there are no nested references to b, replacing b[i] by (b,t:e)[i] (as definition (2.4) does), is the same as replacing b[i] by (*if* i=t *then* e *else* b[i]), we rewrite definition (2.4) as

(3.2) wp("b[r]:=e", R) =

$$\left[R^{b[s1],\ldots,\qquad b[sn]}_{(\underline{\text{if}}\ t=s1\ \underline{\text{then}}\ e\ \underline{\text{else}}\ b[s1]),\ldots,(\underline{\text{if}}\ t=sn\ \underline{\text{then}}\ e\ \underline{\text{else}}\ b[sn])}\right]^{t}_{r}$$

(if the constraint (3.1) holds). This has [illegible] written as a two-stage replacement to emphasize the fact that t refers to the value of subscript r *before* execution of the assignment, while the si refer to the values *after* assignment.

For a postcondition R which is a function of only one or two subscripted variables, R≡R(b[s1]) or R≡R(b[s1],b[s2]), we can further transform (3.2) to arrive at the following two simpler formulas. In them, we assume R has the form

R ≡ R1 *with* t1=v1,...,tm=vm
  ≡ R1 *with* Q (say)

$$(3.3)\quad wp(\text{"b[r]:=e"}, R(b[s1])) \equiv (r\neq s1 \ \underline{and}\ R)\ or\ (r=s1\ \underline{and}\ R1_e^{b[s1]}\ \underline{with}\ Q_e^{b[s1]})$$

$$wp(\text{"b[s]:=e"}, R(b[s1],b[s2])) \equiv$$

$$\begin{aligned}&(r\neq s1\ \underline{and}\ r\neq s2\ and\ R)\ \underline{or}\\ &(r\neq s1\ \underline{and}\ r=s2\ \underline{and}\ R1_e^{b[s2]}\ \underline{with}\ Q_e^{b[s2]})\ \underline{or}\\ &(r=s1\ \underline{and}\ r\neq s2\ \underline{and}\ R1_e^{b[s1]}\ \underline{with}\ Q_e^{b[s1]})\ \underline{or}\\ &(r=s1\ \underline{and}\ r=s2\ \underline{and}\ R1_{e,\ \ e}^{b[s1],b[s2]}\ \underline{with}\ Q_{e,\ \ e}^{b[s1],b[s2]})\end{aligned}$$

That (3.3) is equivalent to (3.2) with n=1 or n=2 can be seen by inspection. For example:

$$\left[R^{b[s1]}_{(\underline{if}\ t=s1\ \underline{then}\ e\ \underline{else}\ b[s1]}\right]^t_r$$

$$\equiv \left[(t\neq s1\ \underline{and}\ R^{b[s1]}_{b[s1]})\ \underline{or}\ (t=s1\ \underline{and}\ R^{b[s1]}_e)\right]^t_r$$

$$\equiv (r\neq s1\ \underline{and}\ R)\ \underline{or}\ (r=s1\ \underline{and}\ R1_e^{b[s1]}\ \underline{with}\ Q_e^{b[s1]})$$

We have written (3.3) with the <u>with</u> clause using R1 <u>with</u> Q instead of R, in order to emphasize that the <u>with</u> clause applies not only to $R1_e^{b[s1]}$, but also to r=s1 (in the second line of (3.3). This is important, because s1 may depend on one of the subexpressions ti=vi in the <u>with</u> clause, and any change in the definition of ti must also bring about a change in s1.

The obvious generalization of this definition to the case where R=R(b[s1],...,b[sn]) for some n≥0, will yield a conjunction of 2^n terms, each of the form.

$$(3.4)\quad (r \overset{=}{\neq} s1\ \underline{and}\ \ldots\ \underline{and}\ r \overset{=}{\neq} sn\ \underline{and}\ R1^{:::}\ \underline{with}\ Q^{:::})$$

We will discuss this and compare it to the original definition (2.4) in a moment, but first let us recompute the weakest precondition for the examples (2.5)-(2.7) using the new definition (3.3). In these succeeding computations, our task is sometimes simplified (cut in half) by the fact that the subscript r in b[r]:=e is identical to one of the subscripts in the postcondition.

(3.5) wp("x[i]:=1", {x[i]=x[j]}) ≡ (i≠j <u>and</u> 1=x[j]) <u>or</u> (i=j <u>and</u> 1=1)
≡ i=j <u>or</u> x[j]=1

(3.6) wp("x[i]:=c1", {x[x[i]]=c2})
≡ wp("x[i]:=c1", {x[t1]=c2 <u>with</u> t1=x[i]})
≡ (i≠t1 <u>and</u> x[t1]=c2 <u>with</u> t1=c1) <u>or</u> (i=t1 <u>and</u> c1=c2 <u>with</u> t1=c1)
≡ (i=c1 <u>and</u> x[c1]=c2) <u>or</u> (i=c1=c2)

(3.7) wp("x[x[j]]:=j", {x[j]=j})
≡ (x[j]≠j <u>and</u> x[j]=j) <u>or</u> (x[j]=j <u>and</u> j=j)
≡ <u>false</u> <u>or</u> x[j]=j
≡ x[j]=j

Let us now compare the two definitions (2.4) and (3.3) (together with its generalization). Definition (2.4), which treats an array b like a function and an assignment b[r]:=e like a change of function, is indeed extremely simple and elegant. However, simplifying a precondition to elide functions of the form (b,r:e) (which is really what we want to do - we want them finally only in terms of b) can be quite complicated. Definition (2.4) allows a simple language definition by hiding the complexity in the manipulation of assertions of the underlying logic.

Definition (3.3) (and its generalization) views an array more as an independent set of (subscripted) variables; an assignment to b[r] changes only that one subscripted variable. Using this definition can be difficult, but further simplification of the precondition is usually easy because no functions like (b,r:e) exist. It is interesting to compare the first derivation of weakest preconditions in examples (2.5)-(2.7) with their later derivations (3.5)-(3.7). For example, one can see the change of weakest precondition in (2.5) from

$1 = (x;i:1)[j]$
to $1 = (\underline{if}\ i=j\ \underline{then}\ 1\ \underline{else}\ x[j])$
to $(i=j\ \underline{and}\ 1=1)\ \underline{or}\ (i\neq j\ \underline{and}\ 1=x[j])$

We made exactly the same kind of transformations in transforming definition (2.4) into (3.3)!

Definition (3.3), which defines b[r]:=e as causing a change in one subscripted variable, brings out one important point. The amount of work that may be required to construct the precondition using the definition can increase tremendously with the number of different references b[si] appearing in the postcondition that may or may not be the same as b[r]. In fact, the amount of work may be as high as proportional to 2^n, for a postcondition containing references b[s1],..., b[sn]. This points out the fact that arrays can greatly increase the complexity of the program unless references to array elements are used in a controlled manner.

Noting the effective similarity between the two array assignment definitions (2.4) and (3.3) (with its generalization) we conjecture that proofs using the original definition (2.4) may also show this

2^n growth in length; the difficulty will of course not be in producing the weakest precondition, but in manipulating it in the hope of obtaining a simpler form, with no functions (b,r:e).

In order to attempt to prove this conjecture, it would be necessary to make explicit the rules of consequence needed to simplify functions references (b,r:e)[i]. We suspect that the two cases (2.4) and (3.3) would just be seen as two extremes in deciding when and how to perform the substitution meant by an assignment and when to manipulate pre and postconditions.

4. The multiple assignment statement

We consider a multiple assignment statement of the form

(4.1) $v1,\ldots,vn,b[r1],\ldots,b[rm] := e1,\ldots,en,f1,\ldots,fm$.

or for short,

(4.2) $\bar{v},\overline{b[r]} := \bar{e},\bar{f}$

where the vi are different simple variables, b is an array, and the ri, ei and fi are expressions. We restrict our attention to assigning to one array only in order to present the idea as simply as possible; the obvious generalization is left to the reader.

In general, we want wp("$\bar{v},\overline{b[r]}:=\bar{e},\bar{f}$", R) to be R with each vi replaced by ei and function b replaced by $(b;\bar{r}:\bar{f})$. However, $(b;\bar{r}:\bar{f})$ may not be a function -- if for example $r1=r2$ but $f1\neq f2$ -- and we must make sure that this does not lead to a precondition that is not well defined. Essentially, we want to develop the weakest precondition such that execution of the assignments in <u>any</u> order will establish the postcondition R. This we do most simply by forming the disjunction of the weakest precondition of all sequences of possible assignments:

(4.3) $$wp(\text{"}\bar{v},\overline{b[r]} := \bar{e},\bar{f}\text{"}, R) = \bigwedge_{\substack{(i1,\ldots,im)\text{ a perm-}\\ \text{utation of }(1,\ldots,m)}} \left(R^{\bar{v},b}_{\bar{e},(b;r_{i1}:f_{i1};\ldots;r_{im}:f_{im})} \right)$$

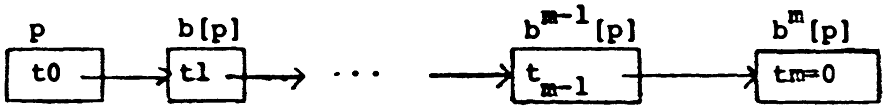
Remember from (2.2) that the function (b;r1:f1;r2:f2) depends on the order of the argument-value pairs listed. This definition is of course a bit difficult to work with formally. It is no problem to construct the precondition, but simplifying it is another matter. In order to simplify the precondition, we use information contained in it together with the fact that functions (b;r1:f1;r2:f2) and (b;r2:f2;r1:f1) differ only when r1=r2 and f1≠f2.

We now illustrate the use of this definition with the following assignment which will be used in the algorithm of Section 5:

(4.4) p,b[p],b[i] := b[p],b[i],p

The postcondition is that p and some of the values of integer array b[1:q] form a linked list ending in 0:

(4.5)



Using the following relation I (which also defines the range of i):

(4.6)	I ≡ f=(t0,...,tm)	[f is the sequence of linked list val
	<u>and</u>(m≥0)	[f contains at least one value]
	<u>and</u>(tm=0)	[The last value of the linked list is
	<u>and</u>(j≠k => tj≠tk)	[All linked list values are different
	<u>and</u>(0≤j≤m => 0≤tj≤q)	[and are 0 or are in the domain of array b.]
	<u>and</u>(1≤i≤q)	

the postcondition is

(4.7) I <u>with</u> t0=p, t1=b[t0],..., tm=b[t_{m-1}].

Note how all references to the variables changed by assignment (4.4) are relegated to the <u>with</u> clause. We want to determine under what precondition execution of assignment (4.4) establishes postcondition (4.7), which is described by picture(4.5). We have:

(4.8) wp((4.4),(4.7)) =
(I <u>with</u> t0=b[p], tj=(b;p:b[i];i:p)[t_{j-1}] for 1≤j≤m)
<u>and</u> (I <u>with</u> t0=b[p], tj=(b;i:p;p:b[i])[t_{j-1}] for 1≤j≤m)

In order to simplify the precondition, we classify it into several cases which characterize just when the precondition might be true, based on the relationship between i, p and the values in list f. Several facts help us here: our knowledge of assertion I (thus for example the fact that all the values ti are different), and the fact that functions (b;p:b[i];i:p) and (b;i:p;p:b[i]) differ only if (i=p <u>and</u> p≠b[i]), and then only for argument i. We have five cases:

(1) i∉f <u>and</u> p∉f. This implies that tj=b[t_{j-1}] for 1≤j≤m.

(2) i∉f <u>and</u> p=tk∈f. p=t0=b[p] leads, with the help of I, to m=0, t0=0 and p=0, so that b[p] would not be defined. Hence, p=tk where 1≤k≤m.

(3) i=tk∈f <u>and</u> p=tj∈f <u>and</u> i≠p. Inspection yields the restriction that k=j+1; otherwise, the value b[p] would occur twice in f, which contridicts assertion I.

(4) i=tk∈f <u>and</u> p=tk∈f <u>and</u> i=p. Because of the restriction 1≤i≤q, we have 1≤k<m. From line 1 of the precondition in (4.8) we have t_{k+1}(tk)=p, and from line 2, t_{k+1}(tk)=b[i]. Hence,

$i=p=b[i]=b[p]$, which together with I implies $m=0$. This contradicts $1\le k<m$. Hence this case cannot arise.

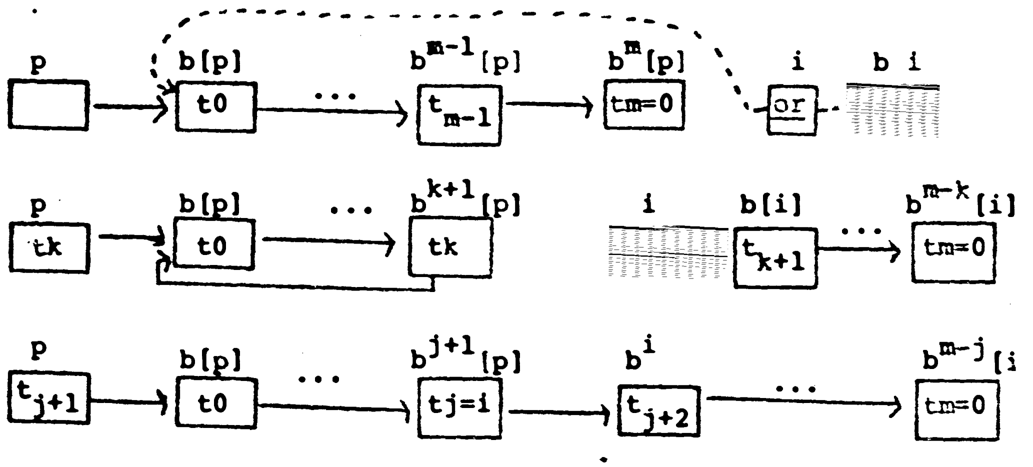
(5) $i=tk\in f$ and $p\notin f$. Here $i\ne p$. From $tm=0$ and $1\le i\le q$ we derive $1\le k<m$. Therefore $t_{k+1}=(b;p{:}b[i];i{:}p)[tk]=p$, which contradicts the fact that $p\notin f$. Hence this case cannot arise.

Hence, we are left with cases 1-3. Rewriting the precondition of (4. in terms of these cases, and simplifying, yields the following precondition (with one line for each case):

(4.9) $(i\notin f \ \underline{\text{and}}\ p\notin f\ \underline{\text{and}}\ I\ \underline{\text{with}}\ f=(b[p],b^2[p],\ldots,b^{m+1}[p]))$
$\quad \underline{\text{or}}\ (i\notin f\ \underline{\text{and}}\ I\ \underline{\text{with}}\ f=(b[p],\ldots,b^{k+1}[p]=p,\ b[i],\ldots,b^{m-k}[i]))$
$\quad \underline{\text{or}}\ (i=a^{j+1}[p]\ \underline{\text{and}}\ I\ \underline{\text{with}}\ f=(b[p],\ldots,b^{j+1}[p]=i,p,b[i],\ldots,b^{m-j}[i]$

We sketch these three cases in (4.10):

(4.10)



Hence, if array b and variables p and i have values which make one of the cases of (4.10) valid, then execution of assignment (4.4) will establish (4.5).

There are several points to make about this example; we will discuss them in the conclusions in Section 6.

5. Proof of correctness of an algorithm

Consider an array c[1:n] of integers, whose sorted order is indicated by a linked list defined by a simple variable p and an integer array b[1:n]. Thus, $c[p]\le c[b[p]]\le c[b^2[p]]\le\ldots\le c[b^{n-1}[p]]$ and $b^n[p]=0$. As an example, we have

n <u>5</u> p <u>4</u>

	1	2	3	4	5
c	23	22	25	21	24
b	5	1	0	2	3

We wish to write an algorithm which sorts array c (changes it to c=(21,22,23,24,25) in this example). The values of p and array b may be changed during execution of the algorithm. Knuth [1973] (exercise 12, pp 81, 596) presents a version of the following <u>linear</u> algorithm due to McClaren:

```
for i:= 1 to n-1 do
  begin   c[p],c[i]:= c[i],c[p];
          p,b[p],b[i]:= b[p],b[i],p;
          while p≤i do p:= b[p]
  end
```

Note that the algorithm uses two multiple assignments, one of which was discussed in detail in Section 4.

Important parts of the invariants of both loops can be divided into three sections, as follows:

(1) Array segment b[1:i] is sorted. Every element in b[1:i] is less than or equal to every element in b[i+1:n].

(2) $f=(p,b[p],\ldots,b^m[p])$ is a linked list ending in 0. It contains the set {i+1:n}. It <u>may</u> contain integers from {1:i}.

(3) Writing f as $f=(t0,t1,\ldots,tm)$, we have $(k<j \text{ and } i+1\le tk,tj\le n)$ $(c[tk]\le c[tj])$.

Using a more formal description,

(5.1) $I(i) \equiv 0\le i<n$ <u>and</u> $c[1:i]$ is sorted <u>and</u> $(1\le j\le i\ \ i+1\le k\le n \Rightarrow c[j]\le c[$
<u>and</u> $(f=(to,\ldots,tm)\wedge m>0)$ <u>and</u> $fm=0)$
<u>and</u> $(0\le j<k\le m \Rightarrow (0\le ti,tk\le n\wedge tj\ne tk))$
<u>and</u> $((0\le k<j\le m \text{ and } i<tk,tj\le n) \Rightarrow c[tk]\le c[tj])$

and using the notation f=linkedlist(p) to mean $f=(p,b[p],\ldots b^m[p])$ for some m, we give in (5.2) a proof outline for the algorithm. Let us discuss the assertions used, which are numbered to the left.

That assertion 1 is initially true is obvious. Next, we have not formally derived precondition (2) from postcondition (3) and the assignment which swaps b[i] and b[p]; we leave this as an exercise for the reader. Note that execution of the assignment puts the smallest value of c[i:n] into c[i], so that I(i) will be true afterwards. Secondly, since these two values are interchanged, we must change the positions of the indices i and p in the list f. All this can be derived using the assignment statement definition given in Section 3, but this is simple enough to be handled informally.
In going from assertion (3) to assertion (4), we see that the value i is deleted from f, or put in a different place if p=i. This is allowed now, since I(i) holds, and therefore c[i] is in the first partition c[1:i]. The list f describes only the ordering of the values in c[i+1:n].

We have derived assertion (4) from assertion (3) because (4) contains exactly two of the conditions under which execution of the next multiple assignment can produce f as a linked list (as described in the last section). Hence we have earlier proved that (5) holds after execution of the second multiple assignment. The rest of the assertions are easy to understand, and no further discussion should be required.

```
(5.2)
  (1)  {I(0)∧p≥1 with f=linkedlist(p)}
       for i:= 1 to n-1 do
  (2)  {I(i-1)∧p≥i with f=(p,b[p],...,b^m[p])}
       c[p],c[i] := c[i],c[p];
  (3)  { (I(i)∧i=p with f=(p,b[p],...b^m[p]))
        ∨(I(i)∧i≠p with f=(i=b^(j+1)[p],b[p],...,b^j[p],p,b[i],...,b^(m-j)[i])) }
  (4)  { (I(i)∧i=p with f=(b[p],...,b^m[p]))
        ∨(I(i)∧i≠p with f=(b[p],...,b^(j+1)[p]=i,p,b[i],...,b^(m-j)[i])) }
        p,b[p],b[i]:= b[p],b[i],p;
  (5)   {I(i) with f=linkedlist(p)}
        while p≤i do {I(i) with f=(b[p],...,b^m[p])}
                     p:= b[p];
                     {I(i) with f=(p,b[p],...,b^(m-1)[p])}
                     {I(i) with f=linkedlist(p)}
        {I(i) p>i with f=linkedlist(p)}
        end
       {I(n-1) with f=linkedlist(p)}
       {c[1:n] is sorted}
```

The algorithm executes in time linear in n. To see this, we note that linked list f begins with n+1 elements. Each execution of the statement p:=b[p] reduces the length of the list by one. Since no statement lengthens the list, the statement p:=b[p] can be executed at most n+1 times in total.

6. Discussion

The first portion of this paper attempted to analyze the conventional assignment to subscripted variables, and to determine where the complexity lay. In general, we feel that viewing an array as a function and an assignment as a change of function is simpler and better than the conventional view of array component assignment. But it should be realized that this does not make problems with arra disappear; it just hides them in the manipulations needed to simplif the precondition of such an assignment. We conjecture that the amou of work needed to simplify such preconditions may indeed be exponent (in the worst case) in the number of different references to array elements that occur in the postcondition. A theoretical analysis of this conjecture is beyond the scope of this paper, but should be performed.

The second portion of this paper introduced a new definition which extends the Hoare-Wirth assignment statement definition to multiple assignment to both simple and subscripted variables. A nontrivial algorithm was given, along with a proof outline which use the new definition.

Although the use of the new definition was quite messy and det we contend that the complexity arose not because of the definition but because of the inherent difficulty of using arrays to implement linked lists and other data structures. Actually, in this case the work involved in finding the precondition wp("p,b[p],b[i]:=b[p],b[i (4.7)) was far less than that required to derive the precondition f the following sequences of assignments which purport to "do the sam thing":

```
t:= p;                        t:=p;
p:= b[t];                     t2:= b[i];
b[t]:= b[i];       or         t3:= b[p];
b[i]:= t                      b[i]:= t;
                              b[t]:= t2;
                              p:= t3
```

The multiple assignment effectively says "reorder the elements of the linked list," while this is harder to see with the above to sequentializations.

It is also true, than one often gets into more trouble dealing with such sequences than one does with the original multiple assignment.

One may of course question the inclusion of multiple assignment as defined by (4.3) in a language, because of the complexity of this definition. This is certainly a point to consider, and one could use the much simpler rule

$$wp(\text{"}\bar{v},\overline{b[\bar{r}]}:=\bar{e},\bar{f}\text{"},R) = R^{\bar{v},b}_{\bar{e},(b;r1:f1;\ldots,;rm:fm)}$$

One must of course not call this a "simultaneous" assignment statement. An implementation, to be both consistent with this rule and efficient, will have to (1) evaluate r1,...,rm,e1,...,en, and f1,...,fm and (2) then store the values in the corresponding variables, but making sure that the values are stored into array variables in the <u>exact</u> order b[r1], b[r2],..., b[rm]. Thus a specific ordering of assignment to the subscripted variables is implied by this definition.

The main point is that the decision of how the statement is to execute should not be made because of intuitive ideas on execution, but instead should be based on being consistent with the axiomatic proof rule we [illegible] to be proper.

The axiomatic definition as applied to assignment (4.4) in the algorithm illustrated a new twist. As programmers, we have always been told that an assignment like p,b[p],b[i]:= b[p],b[i],p is unambiguous and useful only if p=i implies that b[i]=p. Yet here we have an algorithm, and a useful one at that, where i=p but b[i]≠p and still the assignment works correctly. The reason is now clear: if i=p before execution of the assignment, then the postcondition never references the element b[i], and hence it doesn't matter what is stored in it!

This axiomatic approach to defining statements has increased my own understanding of assignment, over what I used to understand via the conventional operational approach. Even such statements as

{true} a[i],a[j]:= 5,6 {a[i]≥5 <u>and</u> a[j]≥5}

make sense now, even if i=j. In a sense, this allows nondeterminism in the same vein as Dijkstra's guarded command structures, although this isn't as important in our theory of programming as his guarded command structures.

One might prefer a simpler definition of assignment in which (for the above example) (p=i <u>and</u> b[i]=p) is invalid. This would require the replacement of p,b[p],b[i];= b[p],b[i],p by

<u>if</u> p=i <u>then</u> p:=b[p]
        <u>else</u> p,b[p],b[i]:= b[p],b[i],p

We contend that the proof of correctness of this statement is just as complicated as the proof of the original one.

Finally, it should be noted that the formal derivation of the weakest precondition brought surprise. Two of the cases for the precondition wp((4.4),(4.7)) = (4.10) were previously known. (see (4.10)); the third came as an unexpected, pleasant surprise, solely from the formal construction.

References

Hoare, C.A.R. An axiomatic approach to computer programming. CACM 12 (Oct. 69), 576-580,583.

Hoare and N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Information 2(1973), 335-355.

Knuth, D.E. The Art of Computer Programming, vol 3. Addison Wesley, 1973.

Dijkstra, E.W. A Discipline of Programming. Prentice Hall, 1976.