PROF. DR. N. WIRTH
ETH
CLAUSIUSSTRASSE 55
CH-8006 ZÜRICH

Array variables.

I have been trained to regard an array in the ALGOL 60 sense as a finite set of elementary, consecutively numbered variables, whose "identifiers" could be "computed". But for two reasons this view does not satisfy me anymore.

The first reason is my abhorrence of variables with undefined values. In the previous chapter we solved this problem by introducing for each variable a passive scope and an active scope, separated by a syntactically recognizable initialization for that variable. But when we regard the array as a collection of (subscripted) variables, that solution breaks down.

The second reason is of a combinatorial nature and more fundamental. In ALGOL 60 the compound statement that causes the variables x and y to interchange their values, needs an additional variable, h say,

h:= x; x:= y; y:= h

which is cumbersome and ugly compared with the concurrent assignment

x, y := y, x .

For the concurrent assignment we have insisted that all variables at the left-hand side should be different: it would be foolish to attach to "x, x := 1, 2" any other meaning than "error". For a long time, however, I hesitated to adopt the concurrent assignment on account of the problems it causes in cases like

A[i], A[j]:= x, y ;

should this be allowed when $i \neq j$, but not when $i = j$? Or is, perhaps, $i = j$ permissible if $x = y$ holds as well, as for instance in

$$A[i],\ A[j] := A[j],\ A[i] \qquad ?$$

If we go that route we are clearly piling one logical patch upon another. However, I have now come to the conclusion that it is not the concurrent assignment, but the notion of the subscripted variable that is to be blamed. In the axiomatic definition of the assignment statement via "substitution of a variable" one cannot afford --as in, I guess, all parts of logic-- any uncertainty as to whether two variables are the same or not.

The moral of the story is that we must regard the array in its entirety as a single variable, a so-called "array variable", in contrast to the "scalar variables" discussed so far. In the following I shall restrict myself to array variables that are the analogon of one-dimensional arrays.

We can regard (the value of) a variable of type "integer" as an integer-valued function without arguments --i.e. defined on a domain consisting of a single, anonymous point--, a function that does not change unless explicitly changed (usually by an assignment). It is somewhat unusual to consider functions without arguments, but we mention the viewpoint for the sake of the analogy. For, similarly we can regard (the value of) a variable of type "integer array" as an integer-valued function of one argument with a domain in the integers, a function, again, that does not change unless explicitly changed.

But the value of a variable of type "integer array" cannot be <u>any</u> integer-valued function defined on a domain in the integers, for I shall restrict myself to such types that, given two variables of that type, we can write an algorithm establishing whether or not the two variables have the same value. If x and y are scalar variables of type "integer", the

this algorithm boils down to the boolean expression x = y , i.e. both functions are evaluated at the only (anonymous) point of their domain and these integer values are then compared. Similarly, if ax and ay are two variables of type "integer array", their values are equal if and only if, as functions, they have the same domain and in each point of the domain their values are equal to each other. In order that all these comparisons are possible, we must restrict ourselves to finite domains. And what is more, besides being finite, the domains must be availabe in one way or another to the algorithm that is to compare the values of the array variables ax and ay.

For practical purposes I shall restrict myself to domains consisting of consecutive integers (when not empty). But even then there are at least two possibilities. In ALGOL 60 the domain is fixed by giving in the declaration --e.g. "boolean array A[1:10], B[1:5]" -- the lower and upper bounds for the subscript value. As a type determines the class of possible values for a variable of that type, we must come to the conclusion that the two arrays A and B in the above example are of different type: A may have 1024 different values, B only 32. In ALGOL 60 we have as many different types "boolean array" as we can have bound pairs (and, as the bound pair may contain expressions, the type is in principle only determined upon block entry). Besides that, the necessary knowledge about the domain must be provided by other means: without further information it is impossible to write in ALGOL 60 an inner block determining whether two global boolean arrays A and B are equal!

The alternative is to introduce only one type "integer array" and only one type "boolean array" and to regard "the domain" as part (aspect)

of any value of such type; we must then be able to extract that aspect from any such value. Let ax be an array variable; in its active scope I propose to extract the bounds of the domain from its value by means of two integer-valued functions, denoted by "ax.lob" and "ax.hib" respectively, with the understanding that the domain of the function "ax(i)" extends over all integers i satisfying

$$ax.lob \leq i \leq ax.hib \quad .$$

Besides those two I propose a third (dependent) one, "ax.dom", equal to the number of points in the domain. The three functions satisfy

$$ax.dom = ax.hib - ax.lob + 1 \geq 0 \quad .$$

(Note that even the empty domain --dom = 0-- has a place along the number line: lob and hib remain defined, they then satisfy hib = lob - 1 .)

We have used here a new notation, the dot as in "ax.lob", "ax.hib" and "ax.dom". The names following the dot are what is called "subordinate to the type of the variable whose name precedes the dot". Following the dot that follows a variable, only names subordinate to the type of that variable may occur and their meaning will be as defined with respect to that type.

Remark 1. In other contexts, i.e. not following the dot, the same names may be used with completely different meaning. We could introduce an array variable named "dom" and in its active scope we could refer to "dom.lob", "dom.hib" and even "dom.dom" ! Such perversities are not recommended and therefore I have tried to find subordinate names that, although of some mnemonic value, are unlikely candidates for introduction by the programmer himself. (End of remark 1.)

Remark 2. A further reason for using the dot notation rather than the function notation --e.g. "dom(ax)" etc.-- is that, unless we introduce different sets of names for these functions defined on boolean arrays and integer arrays respectively --which would be awkward-- we are forced to introduce functions of an argument that may be of more than one type, something I would like to avoid as long as possible. (End of remark 2.)

Remark 3. The expression "ax(i)" is used to denote the function value in point i. Only when the value of "ax(i)" is required needs the argument i to be defined and to satisfy

$$ax.lob \leq i \leq ax.hib \quad .$$

In view of the dot notation we could regard "ax(i)" as an abbreviation for "ax.val(i)", where "val" is the subordinate name indicating evaluation in the point as indicated by the value of the further argument i. For each type, such an abbreviation can be introduced just once! Note that also the type "integer" could have a subordinate name "val", that would enable us to write a little bit more explicitly:

```
x:= y.val
```

instead of the usual and somewhat sloppy x:= y .(End of remark 3.)

For the sake of convenience we introduce two further functions; for the array variable ax they are defined if $ax.dom > 0$. They are

ax.low , defined to be equal to ax(ax.lob) and

ax.high , defined to be equal to ax(ax.hib) .

They denote the function values at the lowest and the highest point of the domain respectively. They are nothing really new, they are defined in terms

of concepts already known and in the definition of the semantics of operations on array values we do not need to mention the effect of them explicitly.

As stated above, a scalar variable can be regarded as a function (without argument) that can be changed by assigning a new value to it: such an assignment destroys the information stored as "its old value" completely. We also need operations to change the value of an array variable --without them it would always be an array constant!-- but the assignment of a new value to it that is totally unrelated to its old value will play a less central role. It is not that the assignment to an array variable presents any logical difficulties --on the contrary, I am tempted to add-- but there is something wrong with its economics. With a large domain size the amount of information stored as "the value of an array variable" can be very large, and neither copying nor destroying such large amounts of information are considered as "nice" operations. On the contrary: in many programming tasks the core of the problem consists of building up an array value gradually, i.e. in a number of steps, each of which can be considered as a "nice" operation, "nice" in the sense that the new value of the array can be regarded as a "pleasant" derivation of its old value. What makes such operations "nice" or "pleasant" depends essentially on two aspects: firstly, the relation between the old and the new value should be mathematically manageable --otherwise the operations are for us too cumbersome to use-- and, secondly, its implementation should not be too expensive for the kind of hardware, that we intend to instruct with our program. The extent to which we are willing to take the latter hardware constraints into account is not a scientific question, but a political one and as a consequence I don't feel obliged to give an elaborate justification of my choices. For the sake of convenience I shall be somewhat more liberal than many programmers

would be --particularly those that are working daily with machinery, the conceptual design of which is ten or more years old--; on the other hand I hope to be sufficiently aware of the possible technical consequences of my choices, that they remain, if not realistic, at least not totally unrealistic.

Our first modification of the value of an array variable, ax say, does not change the domain size, nor the set of function values, nor their order, it only shifts the domain over a number of places, k say, upwards along the number line. (If $k < 0$ it is a shift over $-k$ places in the other direction, if $k = 0$ it is the identity transformation, semantically equivalent to "skip".) We denote it by

ax:shift(k) .

Here we have introduced the colon ":". Its lowest dot indicates in the usual manner that the following name is subordinate to the type of the variable mentioned to its left; the upper dot is just an embellishment (inspired by the assignment operator ":="), indicating that the value of the variable mentioned to its left is subject to redefinition.

Immediately we are confronted with the question, whether we can give an axiomatic definition of the predicate transformer wp("ax:shift(E)", R) . Well, it must be a predicate transformer similar to the one of the axiom of assignment to a scalar variable, but more complicated --and this will be true as well for all the other modifiers of array values-- because the value of a scalar value is fully defined by one (elementary) value, while the value of an array variable involves the domain itself and a function value for all points of the domain. Because the value of the array variable

ax is fully determined by

the value of ax.lob ,

the value of ax.dom and

the value of ax(i) for ax.lob $\leq$ i $<$ ax.lob + ax.dom ,

we can --in principle, at least-- restrict ourselves to post-conditions R referring to the array value only in terms of "ax.lob", "ax.dom" and "ax(arg)" where "arg" may be any integer-valued expression. For such a post-condition R the corresponding weakest pre-condition

wp("ax:shift(E)", R)

is derived from R by simultaneously replacing

1) all occurrences of ax.lob by (ax.lob + (E)) and

2) all occurrences of (sub)expressions of the form ax(arg) by ax((arg) - (E)) .

Note. If E itself depends on the value of ax , the safest way is to evaluate first for the given R with a completely new name, K say, wp("ax:shift(K)", R) , in which then the actual expression E is substituted for K . We have already encountered the same complication when applying the axiom of assignment to statements such as x:= x + f(x). (End of note.)

We give a few examples. Let R be ax.lob = 10 , then

wp("ax:shift(ax.lob)", R) = (ax.lob + ax.lob = 10)

= (ax.lob = 5) .

Let R be (A i: 0 $\leq$ i $<$ ax.dom: ax(ax.lob + i) = i) , then

wp("ax:shift(7)", R) = (A i: 0 $\leq$ i $<$ ax.dom: ax(ax.lob + 7 + i - 7) = i)

$$= R \quad .$$

An alternative way of formulating the weakest pre-condition is

$$wp(\text{"ax:shift(E)"}, R) = R_{ax' \rightarrow ax}$$

(i.e. a copy of R, in which every occurrence of ax is replaced by ax'), where

$$ax'.lob = ax.lob + E$$

$$ax'.dom = ax.dom$$

$$ax'(arg) = ax(arg - E) \quad \text{for any value of } arg \ .$$

From these three definitions it follows that

$$ax'.hib = ax.hib + E$$

$$ax'.low = ax.low$$

$$ax'.high = ax.high \ .$$

Note. Such equalities are meant to imply that if the right-hand side is undefined, the left-hand side is so as well.

For the definition of our further operators we shall follow the latter technique: it describes more clearly how the final value ax' depends on the initial value ax.

The next operators extend the domain at either the high or the low end with one point. The function value in the new point is given as parameter which must be of the so-called "base type" of the array, i.e. boolean for a boolean array, etc. The operators are of the form

$$ax:hiext(x) \quad \text{or} \quad ax:loext(x) \ .$$

The semantic definition of hiext is given by

$$wp(\text{"ax:hiext(x)"}, R) = R_{ax' \rightarrow ax}$$

where
$$ax'.lob = ax.lob$$
$$ax'.hib = ax.hib + 1$$
$$ax'.dom = ax.dom + 1$$
$$ax'(arg) = x \quad \text{for} \quad arg = ax.hib + 1$$
$$= ax(arg) \quad \text{for} \quad arg \neq ax.hib + 1 \quad .$$

The semantic definition of loext is given by

$$wp(\text{"ax:loext(x)"}, R) = R_{ax' \rightarrow ax}$$

where
$$ax'.lob = ax.lob - 1$$
$$ax'.hib = ax.hib$$
$$ax'.dom = ax.dom + 1$$
$$ax'(arg) = x \quad \text{for} \quad arg = ax.lob - 1$$
$$= ax(arg) \quad \text{for} \quad arg \neq ax.lob - 1 \quad .$$

Note. Our earlier remark that also the empty domain would have its place along the number line was to ensure that the extension operators hiext and loext are also defined when applied to an array variable with $dom = 0$. (End of note.)

The next two operators remove a point from the domain at either the high or the low end. They are only defined when initially $dom > 0$ holds for the array to which they are applied; when applied to an array with $dom = 0$, they lead to abortion. They destroy information in the sense that one of the function values gets lost.

The semantic definition of hirem is given by

$$wp(\text{"ax:hirem"}, R) = (ax.dom > 0 \ \underline{\text{and}} \ R_{ax' \rightarrow ax})$$

where $ax'.lob = ax.lob$

$ax'.hib = ax.hib - 1$

$ax'.dom = ax.dom - 1$

$ax'(arg) = \text{undefined} \quad \text{for} \quad arg = ax.hib$

$\qquad\quad = ax(arg) \quad \text{for} \quad arg \neq ax.hib$.

The semantic definition of lorem is given by

$$wp(\text{"ax:lorem"}, R) = (ax.dom > 0 \ \underline{\text{and}} \ R_{ax' \rightarrow ax})$$

where $ax'.lob = ax.lob + 1$

$ax'.hib = ax.hib$

$ax'.dom = ax.dom - 1$

$ax'(arg) = \text{undefined} \quad \text{for} \quad arg = ax.lob$

$\qquad\quad = ax(arg) \quad \text{for} \quad arg \neq ax.lob$.

For the sake of convenience we introduce two further operations, the semantics of which can be expressed in terms of the functions and operations already introduced: they are

x, ax:hipop , semantically equivalent to "x:= ax.high; ax:hirem" and

x, ax:lopop , semantically equivalent to "x:= ax.low; ax:lorem" .

They are given in a notation which is reminiscent of the one for the concurrent assignment; the name following the ":" must be subordinate to the type of the variable immediately before the ":". Obviously, the other variable x must be of the base type of the array variable ax .

The above modifiers all change the domain of the function, either only its place along the number line or also its size. Two further modifiers will be introduced, modifiers that leave the domain as it stands, but only affect one or two function values.

A very important one does not introduce new function values, but only rearranges them. It is of the form

ax:swap(i, j) .

It leads to abortion when invoked without both i and j lying in the domain. Its semantics are given by

$$wp(\text{"ax:swap(i, j)"}, R) = (ax.lob \le i \le ax.hib \ \underline{\text{and}} \ ax.lob \le j \le ax.hib \ \underline{\text{and}} \ R_{ax' \to ax})$$

where

$$ax'.lob = ax.lob$$

$$ax'.hib = ax.hib$$

$$ax'.dom = ax.dom$$

$$ax'(arg) = ax(j) \quad \text{for } arg = i$$

$$= ax(i) \quad \text{for } arg = j$$

$$= ax(arg) \quad \text{for } arg \neq i \ \underline{\text{and}} \ arg \neq j \ .$$

Note. Initially $i \neq j$ is not required: if initially $i = j$ holds, the value of the array variable remains unaffected. (End of note.)

Our last modifier redefines a single function value; it is of the form

ax:alt(i, x) .

It leads to abortion when invoked without i lying in the domain; the

second parameter x must be of the array variable's base type. Its semantics are given by

$$wp(\text{"ax:alt(i, x)"}, R) = (ax.lob \leq i \leq ax.hib \ \underline{\text{and}} \ R_{ax' \rightarrow ax})$$

where

$$ax'.lob = ax.lob$$

$$ax'.hib = ax.hib$$

$$ax'.dom = ax.dom$$

$$ax'(arg) = x \qquad \text{for } arg = i$$

$$= ax(arg) \qquad \text{for } arg \neq i \quad .$$

The operation denoted above as "ax:alt(i, x)" is semantically equivalent to what FORTRAN or ALGOL 60 programmers know as "the assignment to a subscripted variable". (They would write "AX(I) = X" and "ax[i]:= x" respectively.) I have introduced this operation in the form "ax:alt(i, x)" in order to stress that such an operation affects the array ax as a whole: two functions with the same domain are different functions if they differ in at least one point of the domain. The "official" --or, if you prefer: "puritan"-- notation "ax:alt(i, x)" is, however, even to my taste too cumbersome and too unfamiliar and I therefore propose --I too have my weaker moments!-- to use instead

ax:(i)= x

a notation which is somewhat shorter, reminiscent of the so much more familiar assignment statement and still reflecting, by its opening "ax:" that we must view it as affecting the array variable ax. (The decision to write "ax:(i)= x" is not much different from the decision to write "ax(i)" instead of the more pompous "ax.val(i)".)

None of the previous operators can be used for initialization: they can only change the value of an array under the assumption that it has already a value, they can only occur in the active scope of the array variable. We have not yet introduced the assignment

ax:= bx ,

a construct that would do the job. I am, however, very hesitant to do so, because in its full generality "assignment of a value" usually implies "copying a value" and if the domain of the function bx is large, this is not to be regarded as a "nice" operation in present technology. Not that I am absolutely unwilling to introduce "unpleasant" operations, but if I do so, I would not like them to appear on paper as innocent ones. A programming language in which "x:= y" should be regarded as "nice", but "ax:= bx" should have to be regarded as "unpleasant" would be misleading; it would at least mislead me. A way out of this dilemma is to admit as the right-hand side of the assignment to an array variable only enumerated constants, e.g. of the form

(< integer > {, < value of the base type >})

such that

bx:= (5, true, true, false, true)

would establish

bx.lob = 5
bx.hib = 8
bx.dom = 4

bx(5) = true
bx(6) = true
bx(7) = false
bx(8) = true .

The consequence of such a restriction is that assignment of or initialization with a value with a large domain size cannot be written down unnoticed. My expectation is that most initializations will be with values with dom = 0 .

A few concluding remarks are in order.

There is, to start with, the question of economics. My basic assumption is that all operations mentioned in this chapter can be performed at roughly the same price. Some assumption of this nature has to be made, for without it the programming task does not make sense. For instance, instead of writing

```
ax:(5)= 7
```

we could have written the inner block

```
begin glovar ax; privar bx;
  if ax.lob ≤ 5 and 5 ≤ ax.hib →
    bx vir int array:= (0);
    do ax.hib ≠ 5 → bx:hiext(ax.high); ax:hirem od;
    ax:hirem; ax:hiext(7);
    do bx.dom ≠ 0 → ax:hiext(bx.high); bx:hirem od
  fi
end
```

but I would like to reject that inner block as a worthy substitute, not so much on account of the length of the text, but on account of its inefficiency. I will not even regard "ax:(5)= 7" as an abbreviation of the above inner block.

With the possible exception of the assignment of an enumerated value I assume in particular the price of all operations independent of the values of the arguments supplied to it: the price of executing ax:shift(k) will be independent of the value of k , the price of executing ax:swap(i, j) will be independent of the values of i and j, etc. With present-day technology these assumptions are not unrealistic.

It is in such considerations that the justification is to be found for my willingness to introduce otherwise superfluous names: we could have restricted ourselves to ax.lob and ax.dom, for whenever we would need ax.hib, we could write

$$ax.lob + ax.dom - 1$$

instead, but that would make the effective use of ax.hib "twice as expensive" as the effective use of ax.lob and our consciousness of this fact could easily twist our thinking. (Worse: it is guaranteed to do so.)

I said that the prices are of the same order of magnitude. What I also mean is "of the same order of magnitude as other things that we consider as primitive". If the array operations were orders of magnitude more expensive than other operations, we would, for instance, find ourselves invited to replace

$$ax{:}swap(i, j)$$

by

$$\underline{if}\ i \neq j \rightarrow swap(i, j)\ [\!]\ i = j \rightarrow skip\ \underline{fi}$$

and very quickly we should need to know both the exact price ratios and a very good estimate for the probability of hitting the case "i = j" in order to be able to decide, whether our replacement of ax:swap(i, j) by the alternative construct is actually an improvement or not. I know of mathematicians who revel in such optimization problems, problems of which they sometimes seem to thing that they constitute the central problems of computer programming. I leave these problems gladly to them if they are happy with them: the operations that we prefer to consider as primitive should not confront us with such conflicts. I like to believe that we have more important problems to worry about.

A final remark about implementation. It is conceivable that upon initialization of the array variable ax some limits are given: a lower limit for ax.lob, or an upper limit for ax.hib or both, or perhaps only an upper limit for ax.dom. If such "hints to the compiler" are included, a wealth of traditional storage management techniques becomes exploitable. I prefer, however, to regard such "hints to the compiler" not as part of the program: they only make (on some equipment!) a cheaper implementation possible, they represent for the implementation the permission (but not the obligation!) to abort a program execution in which such a stated limit is exceeded.