On the abolishment of the subscripted variable.

(The following is written after discussions with C.Ligtmans, W.H.J. Feijen, M.Rem and C.S.Scholten, while during the first part A.Martin is looking over my shoulder.)

I have been trained to regard an array in the ALGOL 60 sense as a finite set of elementary variables, whose "identifiers" could be "computed". For two reasons this view does not satisfy me anymore and do I prefer to regard an array as a single variable.

The one reason is to be found in my abhorrence for variables, whose values are undefined, a state of affairs caused by the simultaneous introduction of all local variables upon block entry, possibly long before they are actually needed. Two would-be remedies have been suggested (and implemented) many times. The one suggestion is to extend the original range with an additional value "undefined" or "nil"; this is the kind of remedy pure mathematicianse tend to invent, but it leads to all sorts of conflicts, exceptions and logical patches. If one decides on a case of bigamy when two different persons have the same spouse, what about two bachelors, married to the same "nobody"? The other suggestion is to allow initialization and, if not specified, by default a normal value will be assigned (say "0" for an integer and "true" for a boolean variable); the one that follows the second suggestion fools himself even worse, because he has even lost the possibility of a run-time check for a common programming error. In the case of scalar variables a meaningful initialization is no problem if the variables are only introduced when needed; with a set of adequate sequencing primitives there need be no problem in saving the static scope concept: at each semicolon there need not exist any doubt as to which variables together build up the current state space and the common programming error referred to above can be caught mechanically prior to execution. This solution breaks down, however, in the case of arrays regarded as a large collection of elementary variables.

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 to the concurrent assignment

x, y:= y, x .

A requirement of the concurrent assignment is of course that the variables on the left-hadn side are all different: no one should care to give "x, x:= 1, 2" a meaning (different from "error"). For a long time I hesitated to adopt the concurrent assignment on account of the problems it causes in a case like

a[i], a[j]:= x, y .

Should it be allowed when $i \neq j$, but never when $i = j$? Or is it permissible in the latter case if also $x = y$? What about

a[i], a[j]:= a[j], a[i] ?

Clearly we are piling on complication 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 for a variable" --as in I guess all parts of logic-- one cannot afford uncertainty as to whether two variables are "the same" or not.

We can regard a variable of type "integer" as an integer-valued function without arguments --and therefore with a domain consisting of a single anonymous point--, a function that does not change unless explicitly changed (usually by means of an assignment).

Restricting ourselves to the analogon of a one-dimensional array, we can similarly regard a variable of type "integer array" as an integer-valued function with one argument with a domain in the integers --a function, again, that does not change unless explcitly changed.

We now take the view that we shall only admit types such that an algorithm can establish whether the values of two variables of the same type are equal or not. In the case of integer variables x and y this can be done by the boolean expression $x = y$, i.e. both functions x and y are evaluated in the only point of their domain and these values are compared.

Given two variables of type "integer array" their values are equal if, as functions, they have the same domain and in each point of the domain both take on the same value. In order to be able to perform these comparisons, we must restrict ourselves to finite domains and it must be possible to extract from the current value of an array variable the currently corresponding domain. For practical purposes we propose to restrict ourselves to domains consisting of consecutive integers.

If "av" is the name of an array variable, I assume a number of functions of its value and operations upon its value defined. They will be denoted by the identifier of the array variable, a dot, and then a reserved name. Unable to make my choice, I have introduced more than the minimum.

Depending on the domain only are the integer functions "av.size", "av.first" and "av.last"; they will always satisfy

$$av.size \geq 0 \quad \text{and} \quad av.last - av.first = av.size - 1 \quad .$$

The corresponding domain for the function $av(k)$ is given by

$$av.first \leq k \leq av.last \quad .$$

The expression "$av(k)$ --which could be regarded as a special purpose abbreviation for "av.val(k)", a kind of abbreviation one can introduce just once!-- is regarded as any other function call with an integer argument transmitted by value: the argument needs only to be defined and within the domain when the evaluation of $av(k)$ is actually required.

As stated above, a scalar variable can be regarded as a function (without arguments) that can be changed by assignment. Similarly we need operations to change the value of an array variable. For practical reasons is do not consider "assignment" of an arbitrary value to an array variable as a primitive operation: of the domain is large this can be a very costly operation. I therefore prefer array values being built up sequentially by "slight modifications".

Slight modifications are extending the domain by one, at either the "high" or the "low" end: in both cases av.size is increased by 1. With the extension at the high end av.last is increased by 1, with extension

at the low end av.first is decreased by 1. For the argument value added to the domain, a value must be supplied simultaneously, it is really the well-known push-operation and with "x" an expression of the appropriate type we could write "av.hipush(x)" and "av.lopush(x)" ; on the intersection of the old and the new domain the value of the function av will be left unaltered. Similarly, with initially $av.size > 0$ and x a variable of the appropriate type, we can envisage the inverse operations "av.hipop(x)" and "av.lopop(x)" .

A further domain changing operation I envisage is a translation, say "av.trans(k)" , which increases av.first and av.last both by k and leaves for $0 \leq i < av.size$ the value of $av(av.first + i)$ unchanged.

The next thing is assignment. Conceptually there is no problem with an assignment of the form av1:= av2, but if that operation is included it should not look so innocent, and something like "av1.ass(av2)" is already much better. The kind of assignment I am willing to write down with ":=" are the ones in which the final size is small because the function values must be enumerated at the righthand side; I could live with a format of the form

$$av := (0: 0, 1, 2, 3)$$

which defines $av.first = 0$, $av.last = 3$, $av.size = 4$ and $av(i) = i$ for $0 \leq i < 4$. Perhaps even this is already to ambitious and should the righthand side be restricted to values with size = 0. Note that in this proposal even the empty domain "has a place": first and last are always defined: we need this for the operation hipush and lopush to be defined. The same values that are allowed in the assignment are allowed upon the obligatory initialization.

The remaining operations to be discussed are the ones that modify the value of av without changing its domain. An obvious candidate is altering just one of the function values: the analogon of the well-known assignment to the subscripted variable av[k]:= E ; in order to avoid confusion I propose a radically different notation for altering just one function value, say "av.alt(k, E)" in order to do justice to our considerations that we change the whole function, a modification that requires two value parameters.

Another operation that I have learned to consider as "fundamental" --whatever that may mean-- is "av.swap(i, j)". Both arguments must lie in the current domain; if they are equal, it is the empty operation, otherwise the function values av(i) and av(j) are interchanged.

Note. It is unwise to regard the swap as a special case of a cyclic rotation over one place in a cycle of length n with n = 2. Let us not introduce --with a variable numer of parameters!-- av.rot(i, j, k) say: first of all no one would be able to remember in which direction the values would be shifted cyclicly, secondly the difficulties when not all arguments are different --an awkward test, by the way!-- are just terrible. It is the total absence of these difficulties that justifies the view of "swap" as a fundamental primitive.

Finally,what about the declaration, what about our old "subscript bounds"? The implied "bound checking" in av(k) , av.alt(k, E) and av.swap(i, j) is of course with respect to the current value of the domain, it is the test av.first $\leq$ k, i, j $\leq$ av.last. For reasons of storage reservation we can still think of supplying a lower and an upper bound as part of the declaration; they are then no more than a lower bound for av.first and an upper bound for av.last respectively. Alternatively, we could give an upper bound for av.size. I can think of

a) no bounds given
b) a lower bound for first
c) an upper bound for last
d) the combination of c) and d)
e) an upper bound for size.

We recognize the unrestricted stacks and the cyclic buffer of limited capacity.

The drastic difference between these bounds and the old array concept of ALGOL 60 is the following (if I understand the ALGOL 60 array and the notion of type correctly). In ALGOL 60 I have always viewed the bounds as a constant attribute of the array, just as constant as "the type of a variable". From that point of view the one-dimensional integer arrays of ALGOL 60 provide for infinity^2 different "types": viz. as many different types as we can define different bound-pairs.

Here it is proposed that all one-dimensional integer arrays are array variables of exactly the same type, while bounds possibly given in the declaration are regarded as hints to the implementation, hints the implementation may exploit or ignore. They are regarded as more than just a hint, they are also a permission --but not an obligation!-- to abort when the domain exceeds the stated limitations. The bounds when stated in the declaration are <u>not</u> an aspect of the value of the array variable.

The above seems clean and sound. I would welcome comments very much, comments on various levels. It may seem clean and sound, but yet there may be logical difficulties hidden somewhere that I have not seen; if someone finds one, I would be immensely grateful. Also comments on notations and names would be very welcome. (It took me very long to find something for the name "alt" which was eventually suggested by Feijen; and to say that I am proud of my "hipush" and "lopop" would violate the truth.)

The use of the dot in this way --it is the first time I do it myself-- is something in which I have reasonable confidence: it gives us the possibility for the introduction of "restrictedly reserved identifiers". I did not state any scope rules, but the idea is that the fact that av is of type "integer array" makes av.swap identify the operation it should identify. With the dot-notation we can make the post-dot identifiers subordinate to the type of the pre-dot identifier. If av1 is of type integer array, and av2 is of type boolean array, there is yet no "poly-morphism" (I think it is called) involved when we compare av1.val(k) and av2.val(k): the one val is an integer function, the other val is a boolean function. Clearer perhaps is the comparison of av1.swap(i, j) and av2.swap(i, j); this seems nicer than swap(av1, i, j) and swap(av2, i, j), where according to common interpretation these would be regarded as two calls of the same library procedure. As said, I have a reasonable confidence in this use of the dot-notation, but any warning for its pitfalls is welcome.

BURROUGHS
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Research Fellow

20th March 1974