Acta Informatica 10, 229–243 (1978)

Acta Informatica

On Removing the Machine from the Language*

Eric C.R. Hehner

Computer Systems Research Group, University of Toronto,
Toronto, Ontario M5S 1A4, Canada

Summary. Many of the problems in programming languages today are a result of the model of computation offered by our machines. Structured programming combats these problems by restricting flow of control and storage references to a relatively safe set of control and data structures. Some current research attempts to aid programming reliability by enforcing such restrictions in the programming language. This paper proposes a simpler view of some basic language elements (variable, assignment, parameter), not based on our machines, that is safe without restriction. Structuring is then defined independently of what is being structured. A single, simple structure, applied to both values and assignments, yields equivalents for the "safe set" of data and control structures.

0. Introduction

The design of many programming languages, even of those that are claimed to be machine-independent, has been based on the architecture of the computers on which they are implemented. Specifically, the concepts of "data storage" and "flow of control" pervade many languages. Unrestricted storage references and jumps of control are well-known to be dangerous programming practices; under the banner of "structured programming", a restricted set of data structures and control structures have been pronounced relatively safe. For reliable programming, we may further restrict our use of these structures. It is the aim of some current language design research to enforce such restrictions.

We present a simpler model of computing, containing neither the notion of storage nor the notion of control flow, such that one is safe to exercise the model freely, without restriction. The (original) LISP language, based on recursive function composition, is an elegant model without storage or control flow: it contains no assignment statement. Our approach is more ALGOL-like: we retain the assignment as our basic statement, but we shall re-interpret it. We shall insist that Hoare's assignment axiom [5] be valid. Before we present our approach, we wish to show the difficulty with current languages.

* This work was supported by the National Research Council of Canada

0001-5903/78/0010/0229/$03.00

1. The Cell Computer

In some programming languages, including COBOL, FORTRAN, ALGOL 68, and PL/I, the concept of a variable is introduced via the concept of a store, or memory, containing cells. A cell can hold a value and is denoted by an identifier, or name. If one is inclined to the phrase "a variable possesses a value and is denoted by a name", then one is identifying "variable" with "cell". If one is inclined to the phrase "a variable denotes a cell that contains a value", then one is identifying "variable" with the "name of a cell". To avoid confusion, we shall temporarily refrain from using the term "variable"; instead we shall use the terms "name" and "cell". By "name", we shall mean any notation used in a program to denote a cell. In any case, the computing picture is quite clear (Fig. 1).

Given some cells, which may or may not be initialised, and their associated names, we compute by following a program of instructions. There are two distinct aspects to these instructions. Aspect 1—change the values: according to the instructions, we repeatedly write new values into the cells. The result of the computation may be the final value in some cell, or in all cells, or a sequence of values that arose during the computation. The cells may be different sizes, or they may even be made of rubber to accommodate some data types (e.g., character strings); for the moment we will use only small integers. Aspect 2—move the names: the instructions may explicitly require us to change the association of names with cells during the computation. Three language constructs that have given rise to this aspect are pointers, arrays, and reference parameters. If P is a pointer, then a notation such as $P\downarrow$ may be used to name one cell one time, and another cell another time. Similarly, if A is an array, then the name $A[i]$ must be moved to a new cell every time the value in the cell named i is changed. A reference parameter in a procedure may denote a different cell each time the procedure is executed.

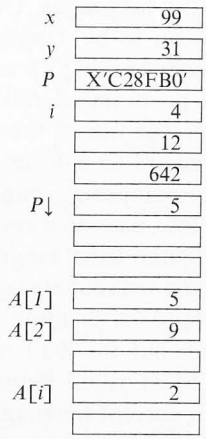

Fig. 1. The cell computer

This "cell computer" poses a danger to reliable programming that has been apparent for several years. An assignment involving one name, such as $x := 5$, may accidentally affect the value indirectly denoted by another name. For example, if a procedure with reference parameters a and b is called with arguments x and x, then an assignment to any one of a or b or x changes all three, since they all denote the same cell. To help the poor programmer, Wirth has proposed a rule: "the necessary prerequisite for being able to think in terms of safely independent variables is, of course, the condition that no part of the store may assume more than a single name" [15]. To ensure that an assignment changes only what it apparently changes, and so make understanding and proving things about programs easier, we may place restrictions on our languages. This means checking that no two arguments to a procedure denote the same or overlapping cells (e.g., $F(A, A[3])$ where A is an array) if either of them may be changed; it means checking that no argument denotes the same cell as a global name used in the procedure, if either of them may be changed; it means ensuring that no two pointers will, at the same time, point to the same cell, perhaps by associating at most one pointer with a data structure. This is the approach taken, for example, by the EUCLID project [9] in its restrictions on PASCAL. Although reliability is greatly increased by prohibiting "sneaky" assignments, the increase is somewhat lessened by the need to understand and remember the restrictions.

2. Two Proposals

This paper puts forward the view that, for reliable programming, two aspects is one aspect too many. Either the names should sit still, or the values should sit still. We shall discuss both of these proposals; the restrictions mentioned in the previous paragraph seem to favour the first proposal, so we shall discuss it first.

Proposal 1. Associate each name in a program with a unique cell.

Pointers are thus eliminated. The suggestion to eliminate them has been made before [6, 7]; the argument has been that they are a data analogy to the **go to** statement, that they do not arise in programs composed in a "structured" fashion, and that their presence in a language invites mischief. We may view their elimination as maintaining the purity of the model: names denote cells; cells contain values. Cells should not contain names. We cannot simply eliminate pointers from some current languages, however, without providing adequate data structuring facilities, any more than we could simply eliminate the **go to** from FORTRAN. Even with good data structuring facilities, those who are used to thinking in terms of pointers will find their loss inconvenient at first, but that is just a matter of training.

In the usual view, an array A is a collection of cells. The assignment $A[i] := 5$ stores the value 5 in the ith cell of the collection. Unfortunately, the array is useful only if the name $A[i]$ denotes different cells at different times, contrary to Proposal 1. The solution is to consider an array as a structured or compound value. The name A may denote a (perhaps large) cell containing an array. The

expression $A[i]$ denotes the ith value in the array; it does not denote a cell, and cannot be used on the left of an assignment. The three-operand expression $A|i \rightarrow 5$ denotes an array like the one in cell A, except that its ith value is 5; it may be assigned to any appropriate cell, such as cell A. (Incidentally, this expression allows us to swap array elements without assignment to a temporary: $A := A|i \rightarrow A[j]|j \rightarrow A[i]$.) This view of arrays has been proposed by Hoare [5] and Dijkstra [1].

An objection may be raised here. Evaluation of an array-expression seems to involve creation of a temporary array. But just as the assignment $i := i+1$ may be implemented as an addition "in place" on a machine with that capability (e.g., PDP-11), so the assignment $A := A|i \rightarrow 5$ may also be implemented as a change "in place" as usual. To broaden the objection, the innocent-looking assignment $A := B$ may hide the complexity of a machine-language loop when A and B are array-valued. However, this same objection may be raised against assignments involving character strings or multiple-precision numbers, depending on the instruction set of the computer, yet we are content with those. Furthermore, in a clever implementation, "lengthy" assignments may simply require changing the address in a descriptor (we must then be careful to change an array "in place" only when one descriptor refers to it; but most of the time, copying is unnecessary). In the end, we must conclude that efficiency can be discussed only in terms of an implementation. But, for the sake of producing reliable software, we cannot allow our language designs to be led by the instruction sets of current computers.

We have discussed the fate of pointers and arrays under Proposal 1. For the third problem, parameters, we defer discussion to Section 9.

Proposal 2. Make the values sit still, and move only the names.

This proposal seems initially less palatable because it does not correspond so closely to an obvious implementation. It becomes instantly more palatable, however, when we realise that it is equivalent to Proposal 1. The cell computer involves two mappings (Fig. 2a). Whether we associate the cells immutably with

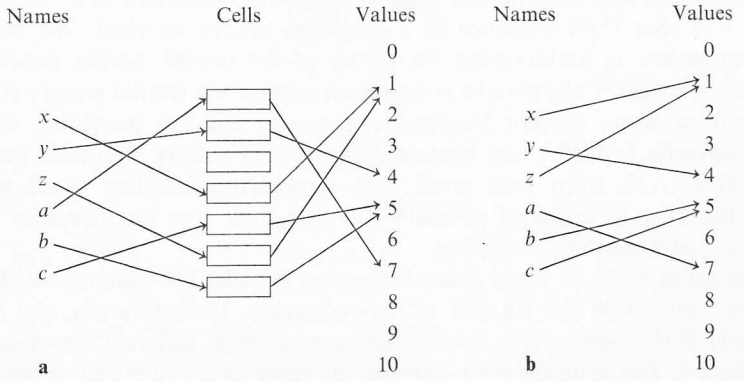

Fig. 2a and b. Mappings. **a** Two-level mapping. **b** One-level mapping

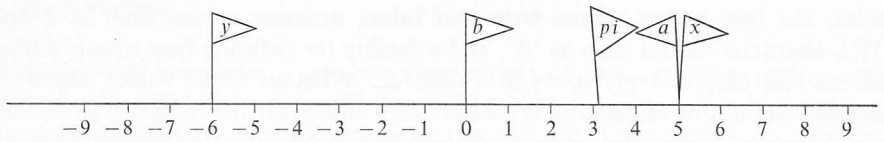

Fig. 3. The flag computer

the names, or immutably with the values, the result is to effectively eliminate one mapping, and make the cells superfluous (Fig. 2b). Donahue [2] has given the mathematical semantics [13] of a portion of a toy language, and shown that the storage domain is required only when one wishes to permit an assignment involving one name to affect another.

In the next section, we present a computing model that is not derived from our machine architecture, and that follows the spirit of the foregoing discussion.

3. The Flag Computer

Consider a number line (or, to generalise to other data types, a space of values). Flags, bearing names, are pinned to various points along the line (Fig. 3). Computing consists of moving flags around according to a program of instructions. For reliable programming, a flag is moved only by an instruction, called an assignment, that explicitly moves the flag. The act of pinning a flag to a value symbolises the act of giving a value a name so that it can be referred to subsequently by that name. At any time, the flag computer is a visual reminder of which names refer to which values.

The model specifies that all flags must be pinned to the line. Inventing pointers would mean allowing flags to be pinned to other flags, so that moving one flag may involve carrying another implicitly.

Similarly, no flag bears the name $A[i]$, for moving flag i would require moving flag $A[i]$ implicitly. Instead, a flag bearing the name A moves in a multi-dimensional space; the expression $A[i]$ projects the flag onto the ith axis; the expression $A|i \rightarrow 5$ describes a point as "the same as A, except that the ith component is 5". Whether an array is considered to be a flag that moves in a multi-dimensional space, the space itself, or a point in the space, does not matter; the reliability criterion is satisfied. For compatibility with the previous discussion of arrays under Proposal 1, we must take "array" to mean a point in space, whose co-ordinates form a "compound value". Equivalently, it is a mapping from a domain of values (specifying the axes) to a range of values (specifying the co-ordinates).

Our purpose in presenting this model is to supplant the cell model, which must be severely restricted to make it safe, with one that is safe without restriction.

4. Naming Notations

In this section, we introduce the elements of a programming language that we shall need subsequently. To begin with, we shall give ourselves some simple

values: the two logical values **true** and **false**, numeric values such as 3 and 3.3E3, character values such as 'A', and a facility for defining new simple values such as *red*, *blue* and *green*, as in PASCAL. Whether these values are truly "simple" will not be debated; it is a convenient choice for the moment. Character strings are not considered simple, and are not included in the above. We give ourselves also a handful of logical, numeric and comparison operators for forming expressions.

We now introduce two notations for giving a value a name. The first gives a value a permanent name. A name is a **constant** if it permanently denotes one value. For example,

pi: 3.14

means that, wherever *pi* appears, it stands for 3.14. A statement of the above form is a **constant definition.** The second notation gives a value a temporary name. A name is a **variable** if it denotes various values, each temporarily. For example,

$x := 5$

means that, wherever x appears, it stands for 5 until some other value is given the name x. A statement of the above form is an **assignment**, and is read "x is assigned to 5". Composition of assignments is denoted by ";" as usual. (Declaration of variables, though important, is irrelevant at present.)

A comment concerning the assignment notation is in order here. Those who would prefer, or even accept, a left-pointing arrow have probably not made the mental adjustment away from the operational semantics of the cell model – the act of storing a value in a cell. If an arrow is used to denote assignment it should be right-pointing to illustrate two things: first, the mathematical semantics – the fact that the program state is mapping from names to values; second, the operational semantics – the act of moving the indicated flag to the indicated value. We have used the right-pointing arrow, however, for another purpose. Our phrase "a name is assigned to a value" comes from Webster's Third New International Dictionary [14], which gives, in its definition of "assign", a most appropriate example: a title (such as "Prince of Wales") is assigned to a person. Several titles may be assigned to one person at the same time; a title may be assigned to at most one person at one time; a title may be reassigned to another person at another time. The current programming usage of the word "assign" (a variable is assigned a value, or a value is assigned to a variable) is contrary to its English usage.

Finally, we introduce a notation for giving an assignment (or a composition of assignments) a name. For example,

increment: $i := i+1$

means that wherever *increment* appears, it will stand for the assignment $i := i+1$. To distinguish *increment* from a conventional label, we emphasize that the above statement does not increment i, but only gives a name to the act of incrementing i.

5. The Structure

Programming languages today provide a variety of data structures (string, array, record) and control structures (**if**, **for**, **while**) that are intended to be convenient for programming. We wish to minimise the number of domains in our language, thus we have not introduced "storage" and we shall not introduce "control". We wish to minimise the number and complexity of the basic axioms describing the language structures, thus we introduce only one simple structure. We define it independently of what is being structured, and apply it to language elements that have already been introduced. We must show that we have not gained simplicity at the expense of convenience, so we shall identify special cases of the general structure that are approximately equivalent to the structures mentioned above.

Our structure is simply a set of ordered pairs. The first member of each pair is called an index, the second an element, of the structure. Indices are values. (It may seem appropriate to restrict the indices of a given structure to one type, or to a contiguous subrange of a type, but such restrictions are irrelevant to this paper.) An expression of an index may include variables, although our examples will use only constants. The elements of a structure constitute that which is being structured; we shall consider structured values and structured assignments.

6. Structured Values

Define a map as a structure whose elements are values. For example, the map

map ⟦ 1→11 | 2→12 | 3→13 ⟧

has indices 1, 2 and 3, and elements respectively 11, 12 and 13. A map is itself a value, and can be used in an expression, or as an element of a map. It may be given a name by the same notations used to give a simple value a name (constant definition, assignment). If M is or denotes a map, and i is or denotes an index of M, then $M[i]$ denotes an element of M corresponding to index i (if this element is to be unique, the indices must be unique). Other operations on maps may be defined as deemed necessary; for example, if M is a map and i and e are values, then the expression $M|i \rightarrow e$ denotes a map like M except that $M[i] = e$.

When the indices are the integers 1 to n (for some n), the notation may be abbreviated by lising only the elements, in order of index, enclosed in angle brackets. The map of the previous paragraph may be expressed as ⟨11, 12, 13⟩. This gives us a kind of array. If the elements are characters, e.g. ⟨'A', 'B', 'C'⟩, we may abbreviate further, e.g. "ABC". This gives us character strings. The equivalent of records (or PL/I structures) are formed, not as an abbreviation, but simply by using suitable programmer-defined values as indices.

```
map ⟦name    →"HEHNER"
    |address →map ⟦city    →"TORONTO"
                  |country →"CANADA"⟧⟧
```

We use the abbreviation n **to** m to stand for $n, n+1, \ldots, m$ whenever n and m denote appropriate values. Thus the array $\langle 11, 12, 13 \rangle$ could be written $\langle 11$ **to** $13 \rangle$. We find it convenient to group indices that have equal elements, so that

$$3, 4, 5 \rightarrow 50$$

or

$$3 \textbf{ to } 5 \rightarrow 50$$

stands for three index-element pairs. One more convenience we mention, for reference later, is the selection of several elements at once. Let $M[i, j, \ldots, k]$ be short for $M[i], M[j], \ldots, M[k]$. So, for example, "ABCDE"[2 **to** 4] means the characters 'B', 'C', 'D' and provides a substring operation.

The full form of a map begins with a declarative line that serves three purposes. It introduces a name for the index, so that we can express elements in terms of their indices. It specifies a range for the index, i.e., a domain for the map; this gives us either redundancy that is useful for checking that all indices are present, or the ability to specify a restricted **else**. And it specifies a range for the map. For example,

map i: 1 **to** 100 $\rightarrow$ 0 **to** 99
$[\![3, 5, 10 \rightarrow 50$
$\mid 11$ **to** $20 \rightarrow i * 2$
$\mid$**else** $\rightarrow 0]\!]$

is a sparse array containing thirteen non-zero and eighty-seven zero elements.

In some cases, a compiler may represent a map as a sequence of pairs of values; in other cases only the elements need storage. In still other cases, maps may be compiled into code for computing values from indices. As an example in which the latter is suitable, consider the following recursive definition of limited factorial.

factorial: **map** n: **integer** $\rightarrow$ **integer**
$[\![0 \rightarrow 1$
$\mid$**else** $\rightarrow n * factorial[n-1]]\!]$

Comparing two maps of this sort for equality is probably not a good idea.

The generalisation to maps of more than one index (multi-dimensional arrays) is straightforward, if we are careful to choose distinctive punctuation. A map of no indices has exactly one element.

The reader may, at this point, feel drowned in notation. But he should not feel drowned in concepts or language features. Different programmers will have different data structuring requirements, and different notations and abbreviations that seem convenient for their purposes. No programming language can hope to provide all such notations, nor should it attempt to. Instead it should provide a general structure, and a mechanism for specialising it to an individual's needs, and for making convenient abbreviations. This paper does not discuss such mechanisms. It suggests that the map can easily be specialised to provide a variety of programmer's needs, while maintaining an economy of concepts within the programming language.

7. Structured Assignments

Define a plan as a structure whose elements are assignments (or compositions of assignments). For example, the plan

plan $[\![1 \to x := 11$
$\quad |2 \to y := 12$
$\quad |3 \to z := 13]\!]$

has indices 1, 2 and 3, and elements respectively $x := 11$, $y := 12$ and $z := 13$. A plan may be given a name by the same notation used to give a simple assignment a name. If P is or denotes a plan, and i is or denotes an index of P, then $P[i]$ denotes an element of P corresponding to index i. (Once again, if the element is to be unique, the indices must be unique. We do not make this restriction, however, and we can build Dijkstra's guarded command sets [1] by using logical expressions as indices.)

Thus one builds a plan and selects elements from it the same way one builds and selects elements from a map. If P is a plan then $P[i]$ is evidently a case statement. An **if** construct is easily defined as a special case. For example,

if b
$\quad$ **then** $x := 5$
$\quad$ **else** $y := 7$ **fi**

abbreviates

plan $[\![$**true** $\to x := 5$
$\quad |$**false** $\to y := 7]\!]$ $[b]$

Similarly, **if** ... **then** ... **fi** abbreviates the special case when the **false** alternative is null.

The full form of a plan begins with a declarative line that serves three purposes. The first two are the same as for maps. The third is to specify the variables that are (re)assigned by the elements of the plan (analogous to the range of a map). We now define a **for** construct as an abbreviation. For example,

for i: 1 **to** 3 **do**
$\quad s := s + i$ **od**

is an abbreviation of

plan i: 1 **to** 3 $\to s$
$\quad [\![1$ **to** $3 \to s := s + i]\!]$ [1 **to** 3]

which means

plan i: 1, 2, 3 $\to s$
$\quad [\![1 \to s := s + 1$
$\quad |2 \to s := s + 2$
$\quad |3 \to s := s + 3]\!]$ [1, 2, 3]

which ultimately means

$s := s+1;$
$s := s+2;$
$s := s+3$

Here we have made several selections within one pair of brackets in the same fashion that give us the "substring" operation for maps. Notice particularly that index names are not variables, so no question of assignment involving indices is raised. The generalisation to more (or less) than one index is, once again, straightforward.

Finally, the notation **while** ... **do** ... **od** may be used to abbreviate a recursively defined plan. For example

while $i < 10$ **do**
    $i := i+1$ **od**

abbreviates a plan, say w, defined as

w: **if** $i < 10$
    then $i := i+1;$ w **fi**

8. Understanding Structured Assignments

When we read or write programs, we have always maintained a thinly disguised "program counter" or "instruction pointer"; we are always aware of the flow of control. For example, at the end of a loop we know that control or execution goes back to the beginning of the loop. We have no problem understanding the **go to**: execution continues at the indicated statement (understanding programs containing **go to**s is another matter). A program should not be understood in terms of a particular implementation of a language, nor by tracing an execution of the program. These two premises are generally well accepted, and need no defense in this paper. Yet in almost every programming text, control structures are explained only by explaining how to trace an execution according to some implementation.

Selection (**if** or **case**) is usually explained by saying that control jumps to one of the alternatives, and from there to the end of the construct. In this paper, the notation $P[i]$ is said to denote an element of P. Apart from the subtle change in attitude, the use of the word "jump" may sometimes be misleading. For example, with the constant definition

devices: 4

the structured assignment

if *devices* < 10
  then A
  else B **fi**

can be compiled simply as A, giving conditional compilation without any new language feature. When the selection is made at compile time, there is no jumping in the execution.

We introduced the **for** construct as an abbreviation for a sequence of assignments, rather than as a "loop". The important point is not the change in terminology, but the change in thinking: from jumping control to structured assignments. One could never invent a **go to** as a structured assignment.

An explanation of recursion that involves activation records or return address stacks is irrelevant, confusing, and often wrong. The best implementation of the recursive plan in the previous section is exactly the same as the implementation of the **while** construct that abbreviates it [8]. The proper explanation of a **while** construct is the same as that of the recursive construct that it (slightly) abbreviates. The complete formal semantics of the **while** (or other unbounded looping) construct, e.g. by weakest-precondition predicate transformer [1], require that it be regarded the same as recursive selection [4]. Viewing **while** as a control construct, i.e., a loop, one may be tempted to invent an intermediate or deep **exit**; the benefit is an efficiency not attainable with only the **while** loop, the detriment is a loss of clarity in the programs [10] and the introduction of a new domain in the formal semantics (continuations). However, by structuring assignments rather than control, we can achieve the same efficiency as with **exit**, with a clarity exceeding that attained by using only **while** [4], without requiring any new domains.

9. Parameters

According to an early FORTRAN compiler, if R is a subroutine with parameter X, such that the invocation $R(3)$ causes execution of the assignment $X=4$ within R, then ever after, 3 should stand for 4. Since then we have learned to distinguish among parameters by value, by name, by reference, by result, and by value-result. The purpose of this section is to rescue from this confusion a simple notion of parameter.

A module is a piece of a program that has the following three characteristics.

(a) Its outside characteristic (net effect when executed/invoked/instantiated) is that of a basic language element.

(b) Inside a module, the full power of the language is available to produce the intended effect, including naming, structuring, and creating modules.

(c) The method of producing the intended effect is hidden from the user (invoker) of the module, i.e. names are local.

One kind of module, called a function, has the outside characteristic of a map, and can be used where a map can be used. For example, it may be given a name, to stand in place of the function, by the notations already introduced. It can be "indexed" by the notation already introduced. (As was the case for some maps, comparing functions for equality is probably not a good idea.)

We do not now introduce the idea of a parameter; we did that in Section 6, except that we called it "a name for an index". We suggest that a parameter of a function should not differ from a parameter of a map. It should stand, inside

the function, for a value in terms of which the result is expressed. Thus it is, in essence, an unknown constant. (We shall defend this view in a moment.) The declarative line at the beginning of a map is appropriate also for a function (this is standard in programming languages).

A routine is a module that has the outside characteristic of a plan, and can be used where a plan can be used. It can be named and "indexed" by the notations already introduced. Here too, a parameter should be an unknown constant. The declarative line of a plan is appropriate for a routine: it specifies, in addition to the name and range of the parameter(s), those variables that may be changed (reassigned) by the routine—and that is the routine's net effect. This list of variables is approximately what is becoming known as an "import list" [9, 16].

Our function returns a value, but does not change global variables; our routine changes global variables, but does not return a value. According to our definition of module, we could invent a module (call it a procedure) that does both only if we were willing to include in the language an element that is both an assignment and a value (as in *APL*). The pros and cons are the same at both levels.

Now we can come to the main point of this section: that we should resist the temptation to embellish the notion of a parameter beyond the simple notion that it stands for a value to be supplied as an argument. Let us examine two such temptations to which we have succumbed in the past.

Perhaps the fact that a parameter stands for different values at different times suggests that it is a kind of variable. But, for a given execution, each argument supplies one particular value for the corresponding parameter. If we reassign the "parameter" to some value other than the argument, then it is no longer acting as a parameter, but as a local variable. If one wishes to invent a local variable, with an initial value that depends on one or more parameters, that is acceptable. But the decision to create a parameter, and the decision to create a variable, should be separate decisions. The possible values of a parameter are part of the specification of a module; the sequence of values of a local variable are part of its implementation. The two should not be confused.

The other temptation is to allow a parameter to stand for more than just a value. For example, it may seem useful to allow a parameter to stand for a variable (either reference or result parameter), so that a routine can affect different variables on different executions. This would detract from reliability in two ways: as argued in an earlier section, names should not stand for names; and, as mentioned a few paragraphs ago, a routine should specify which variables it affects. At first sight, this reliability criterion may seem too strict, for it prevents us from writing a sort routine capable of sorting more than one specific array. We can, however, write a sort function, i.e. $A := sort[A]$ or even $B := sort[A]$, and this requires no embellishment of the simple notion of a parameter.

To generate efficient code for the above assignments, a compiler must realise two things. First, assuming that it has allocated space for the compound values denoted by each of A and B, it can develop the result of the function in the space allocated to A in the first assignment, and to B in the second. There is no need for temporary storage or recopying of the result. The mechanism needed is exactly the same as would be needed for a "result parameter". Second, if the

parameter of *sort* is used only to initialise the result variable, then the argument may be copied directly into the space allocated for the result. In the first assignment above, that means copying from A to A, which needn't be done at all; thus we have a "reference parameter" implementation. Both of these compiler activities are more generally applicable than just to the above situation. The first is required for assignments such as $i:=i+1$ and $A:=A|i\rightarrow 5$ as discussed previously. The second is applicable whenever any constant is used only to initialise variables. But all of the above considerations are just the details of some imagined implementation, and language design should be independent of the details of any particular implementation.

An example commonly used to justify "parametric procedures" in ALGOL or PL/I is the need to pass an integrand to an integrate procedure. Since a function, as defined above, is allowable where a value is allowed, we satisfy this need without complicating parameters, and without the horrors of side-effects changing distant environments. If we are faced with a compelling example, we will have to consider allowing parameters to stand for more than just values. Such an example must not be a programming situation that, once we are in it, is best gotten out of by means of complicated parameters; rather, it must be a problem that is best solved by means of complicated parameters. Until we find that such examples are common, we should maintain the simplest possible view of parameters.

10. Uniform Referents

When programming, if one decides that one needs to select the ith value from a class A of values, one uses the notation $A[i]$. One may then decide to implement A either as a map, or, if one needs to introduce local names and to use assignments to produce the desired value, as a function. The notation $A[i]$ does not prejudice the choice of implementation of A. This principle of language design is known as "uniform referents" [12].

We now have a new application for the principle. If one decides that one needs to select the ith assignment (or composition of assignments) from a class S of assignments, one uses the notation $S[i]$. This allows S to be implemented as a plan, giving us a "case statement", or as a routine. A change in the implementation of S that does not change the meaning of the program, such as substituting a plan for a routine or vice versa, should not require changing the notations involving S throughout the program.

11. Conclusion

The common view of variables, as they appear in currently popular programming languages, comes from our history of machine-directed language design. Storage cells, and their addresses, have their exact counterparts in the so-called "high-level" languages. An implementer of programming languages must be concerned with storage cells, with address calculations, and with load and store instructions.

But to a user of programming languages, a different view may be more appropriate. We have suggested that a constant definition and an assignment should both be considered simply as giving a value a name, in the first case a permanent name, in the second case a temporary name, so that it can be referred to subsequently by that name. It is important that the name denote the value directly. The introduction of storage cells, such that a name denotes a cell that contains a value, adds a level of indirection that impairs our ability to program reliably, and tempts language designers to invent harmful things such as pointers and reference parameters. It has subverted our view of arrays from the mathematical structure (a structure of values, or a mapping from values to values) to a storage structure. To a seasoned programmer, the cell model is so familiar that abandoning it is difficult, even when its shortcomings are recognised. To aid in the transition, we have proposed another model: the flag computer. But naming is familiar to everyone; people are named, cities are named, objects are named. To a new programmer, the act of assigning a name to a value scarcely needs an explanation or a model.

Flow of control, like the storage reference, is a machine-level concept that requires restriction for safety in high-level languages. But restrictions are often resented, and seldom understood. By structuring assignments, rather than control, we create a language that is not perceived as restrictive: we cannot create things like **go to** and **exit** that complicate the mathematical semantics and obscure our understanding. Structured assignments are more conducive to a mathematical style of program composition, while (structured) control is more conducive to tracing executions.

We do not defend the notations used in this paper; when the reader finds them awkward or unpleasant he is invited to change them. Nor do we suggest that the particular structure we have chosen for structuring values and assignments is adequate or suitable for all purposes. Our point is that structuring can be defined independently of what is being structured, and then applied profitably to more than one domain. A well-chosen general structure that can be specialised in a variety of ways is preferable to the myriad of seemingly unrelated structures we now live with.

ALGOL-like languages introduce, all in one complicated package called the "procedure", the ability to name a refinement (and so use the name in its place), recursion, parameterisation, and local scope. And the "parameter" is often a complicated combination of parameter, local variable, and result. This has caused unnecessary confusion for programmers and headaches for implementers. We introduced naming (which allows recursion), parameterisation (i.e. the structure), and local scope (i.e., the module) separately. We took the simplest view of parameter: a constant that lacks local definition. What we have traditionally called "binding by value", "binding by name" and "binding by reference" become implementation alternatives under some circumstances; they are removed from the concerns of the programmer.

Acknowledgements. I am indebted to Jim Horning and Nigel Horspool for discussions that led to this paper.

References

1. Dijkstra, E.W.: A discipline of programming. New Jersey: Prentice-Hall 1976
2. Donahue, J.E.: Locations considered unnecessary. Acta Informat. **8**, 221–242 (1977)
3. Hardgrave, W.T.: Positional versus keyword parameter communication in programming languages. SIGPLAN **11**, 52–58 (1976)
4. Hehner, E.C.R.: **do** considered **od**: a contribution to the programming calculus. CSRG Technical Report 75, University of Toronto, 1976
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–583 (1969)
6. Hoare, C.A.R.: Recursive data structures. Internat. J. Comput. Information Sci. **4**, 105–132 (1975)
7. Kieburtz, R.B.: Programming without pointer variables. ACM Conference on Data: Abstraction, Definition and Structure, Salt Lake City, 1976
8. Knuth, D.E.: Structured programming with **go to** statements. ACM Computing Surveys, 6.4, December, 1974
9. Lampson, B.W., Horning, J.J., London, R.W., Mitchell, J.G., Popek, G.J.: Report on the programming language Euclid. SIGPLAN **12**, 1–79 (1977)
10. Ledgard, H.F., Marcotty, M.: A genealogy of control structures. Comm. ACM **18**, 629–639 (1975)
11. McKeeman, W.M.: An approach to computer language design. Report STAN-CS-66-48, Stanford University, 1966
12. Ross, D.T.: Uniform referents: an essential property for a software engineering language. In: Software engineering (J.T. Tou, ed.). New York-London: Academic Press 1970
13. Scott, D., Strachey, C.: Towards a mathematical semantics for computer languages. In: Computers and automata (J. Fox, ed.), pp. 19–46. New York: Wiley 1972
14. Webster's Third New International Dictionary, p. 132. Springfield, Mass.: Merriam 1976
15. Wirth, N.: On the design of programming languages. Proc. IFIP Congress, Stockholm, 1974. In: Information processing, Vol. 74, pp. 386–393. Amsterdam: North-Holland 1974
16. Wirth, N.: Modula: a language for modular multiprogramming. Technical Report 19, Institut für Informatik, Eidgenössische Technische Hochschule, Zürich, June, 1976

Received July 20, 1977