On representational abstraction.

The usual higher level programming languages provide us with the type "integer" allowing us to prescribe "a:= b + c" or to assert "a = b + c", regardless of the internally chosen interpretation. This is great, for the alternative would have been to regard these variables as bit strings, say; this would have been nasty in two respects: instead of asserting "a = b + c" we would find ourselves forced to write down the equations of the binary adder --which on that level would only add to the confusion--, secondly we would be overspecific, because we would (presumably) not object if our program operating on integers would be executed on a decimal machine! So much for the power of representational abstraction. Its power is so great that we would like to exploit it at higher levels as well. We should, however, be aware of one vital fact, viz. that while the highest level program does not fix a representation for variables of type "integer", it is fixed somewhere anyhow, viz. in the implementation: someone has to bother!

The most innocent way of regarding "a type" is as "an attribute of variables of that type", i.e. as the collection of different possible values a variable of that type can have. (How this collection of different values is defined is another matter: either by enumeration or by some BNF-like recursive technique I guess.)

We should be aware of the fact that this abstract definition of the set of possible values is only one side of the coin: to compare it with the procedure concept --embodying operational abstraction-- it is as if we have given an exiomatic definition of the semantics of the call without --as yet-- specifying an adequate body.

To do justice to this observation we could talk for the purpose of this discussion about "an implemented type"i.e. a defined class of values together with a proposal as how to represent these different values. In addition it should contain a proposal for the algorithmic operations on the consituents of the representation in order to "model" the operations on the abstract values.

In the simplest case this proposal associates with each abstract value one or more points in a little state space which is probably regarded as the Cartesian product of smaller --"known"-- state spaces. (One in the case of "a unique representation", more in the case of a "non-unique representation". Note that not all points in our little Cartesian state space need to correspond to a possible value of the abstract type!) In any case our proposal for representation needs to give the correspondence between abstract values and points in the little state space.

On the above level of vagueness, everything is still OK: we have not said much yet that could contradict itself. Let us, therefore, look for problems.

There seems to be a marked difference between operational abstraction and representational abstraction in the following sense. Given a program with procedures the replacement of one procedure body by a semantically equivalent one is a straightforward operation: the choice of the body is only reflected in the "static) program text and in the (volatile!) happening when the program is executed: with modest precautions it ~~is~~ even possible to replace during execution of a program one body by a semantically equivalent one that will be exercised during the second half of the computation.

The choice of representation, however, spreads all through the state space of the computation: to start the execution of a program in binary arithmetic and to switch half-way to decimal arithmetic is certainly <u>not</u> such a trivial operation. One can even start asking oneself, whether one could have both representations simultaneously: could we have simultaneously "binary integers" and "decimal integers"? The fact that there are machines with both binary and decimal arithmetic --at least: machines that are presented as such-- seems to make this question meaningful. It is, however, utterly misleading.

There are no such things as "binary integers", nor "decimal integers". Both names are misnomers! The integers, according to Kronecker created by the Good Lord and subsequently defined by the axioms of Peano are a concept in which "a number system" has no place, let alone a specific number system.

We can have as types (possibly bounded) integers, and/or bitstrings and/or decimalstrings. But neither a string of bits, not a string of decimals can be an integer. We can associate --but this is at least a conceptual transfer function!-- with either string an integer value, but in both cases that requires a quite distinct additional convention.

(In this connection it is illuminating that the definition of a programming language like ALGOL 60 does not mention a thing about internal representation of integer variables, this in sharp contrast to most machine manuals that usually start with ... a detailed description of the number representation, the last thing that should be of any importance for the programmer! Instead of operating on integers, one has there to operate on the constituents of a specific representation and that is, of course, one of the reasons why machine code programming is so tricky: you really have to know the tules for representing negative numbers on a binary machine before you can translate the test "even(x)" into an inspection of x's least significant digit. As an aside: from previous confrontations with COBOL I have retained the impression of an endless conceptual confusion between integers and strings of decimal digits ....)

Similarly there is no "binary arithmetic" nor "decimal arithmetic": Peano did not mention radix systems in his axioms!

Let us now assume for a moment that we have a binary machine with a number of "decimal operations" as an added facility; how are we going to use it? (Note 1. It will become apparent that such an added facility is to a large extent an added nuisance --as usual?-- but we shall investigate its use nevertheless. Note 2. The dual number representation is a rather microscopic example of non-unique representation: for more sophisticated types than integers it becomes more interesting.) In a purely binary machine our higher level code does know nothing about the number representation; in our mixed machine we should stick to that same level of transparancy. I.e. if an integer value can be represented by a string of bits or a string of decimals (or possibly both) we should have what on that level is called "a tag", indicating which is the case. (A proposal could be something on the following lines:

1) if both operands are available in one kind only and the kinds are equal, produce the result of the same kind

2) if both operands are available in one kind only and the kinds differ perform concurrently bot transfer functions and then act as

3) if both operands are available in both kinds, produce concurrently the result of both kinds

4) if one operand is available in both types, the other one in one kind only, produce a result of the latter kind, etc.

The above is not a serious proposal, it is intended as an example how the implementation could try to choose at any moment the most convenient alternative.)

It could very well be that the programmers knows that for some variables a specific representation is more adequate than another. (This is particularly true when you have variables of the type "set".) But it is my feeling that such knowledge should be transmitted to the implementation in the form of "a hint": a parameter of the declaration that introduces the variable. It is them up to the implementer to decide how much attention will be paid to the hint. (In our example: even with hints of the form "integer (dec)" and "integer(bin)" a program executed on the machine with the decimal facilities should be transferrable to a machine without the decimal facilities without any change --the purely binary machine implementation can be expected to ignore hints to do something in a locally impossible way!

One can argue with respect to such a text with hints: "Is the programmer now aware of the representation or is he not?" Well, both of course! He creates the major part of his text as if he could ignore the representation, but he knows full well that if such an abstraction is insufficiently truthful, this full separation of concerns is denied to him and he has to develop a controlled schizophrenia. That is what all programmers sometimes are forced to do: we only try to assist him in controlling his schizophrenia. What we try to achieve is that, although the program is perhaps only realistic when the "hints" are taken into account, yet the program without the "hints" can be regarded as "logically complete".

* * *

For the sequal I take the position that we aim at a strict textual separation between the places where the abstract variable is considered as an unanalyzed whole and where it is considered as a composite object. Note that this is meant to be a drastic decision, for instance damning ALGOL 60 --and many other programming languages-- where on the same level one can equally well refer to an array as a whole as to its individual elements, allowing procedure calls such as "P(A, A[i])", which I would prefer to consider as confusing horrors.

If we take such an attitude, we should pay attention to the following question --even if we do not have complete and completely satisfactory answers to them:

a) What are exactly the benefits that we hope to derive from such a separation?

b) What type of interface do we suggest between the two levels?

A clear benefit is presented by the circumstance that --provided that the abstract variable has decent properties-- the upper level program can be much clearer to understand and --even formally-- easier to prove to be correct. A program operating on complex numbers becomes mystified when you express it in real and imaginary parts! (Most formal proof systems that have been mechanized --King's "Verifying Compiler" and the like-- seem to suffer from the fact that they carry out the proofs in terms of the primitive data types handled: the proofs become very quickly unwieldy.)

A second benefit arises as soon as not all points of the local little state space --built up as the Cartesian product of the state spaces of the components-- actually correspond to a value of the abstract variable. In that case there is redundancy, a relation in that little state space must be kept invariant. The textual separation pins the obligation to maintain such an invariance quite explicitly down on that part of the text that manipulates on the individual components. As many program improvements boil down to a trading of storage space for computation time by storing redundant information --that might be absorbed in the representation of an abstract type?--, this second benefit also seems to be important. (It is here where the "unsafeness" of for instance the PASCAL record really begins to hurt!)

As the third benefit I see the possibility to change the representation of an abstract type: in a sense the language "scope rules" will tell you which parts of the program remain unaffected.

There is a fourth potential benefit of which I am not quite sure: it might be to far-fetched, but you never know in these days of privacy and security. It is conceivable that we would like to be able to give partial access to information only; and mind you, this will not necessarily be that certain bits are inaccessible and therefore unable to influence the course of the computation: while the "age of a person" might be hidden, you may yet get permission to compare the ages of two persons: the relational operator "older than" might be at your disposal! I think that more convincing examples can be constructed....

Let us now turn to the question b), what type of interface we do suggest between the two levels.

We cannot only define an abstract variable, its layout in store, so to speak, we must define operations and functions as well. In the case that the abstract variable is represented by an enumerated set of variables --such as you can declare at block entry-- it has been suggested by C.A.R. Hoare --I cannot give the exact reference, I think it was ACTA INFORMATICA-- to write a block with

a) the declarations of the components needed to represent an instance of the abstract variable, together with their initialization

b) functions defined on a variable of such a type: the algorithm would then refer to the components of the abstract variable

c) updaters, i.e. operations modifying the value of the abstract variable.

In the case of, for instance, the complex number, there would be certainly the two functions "re" and "im". They can be regarded as "characteristic functions" in the sense that if you know these two values you know all there is to know about that complex value. If "z1" and "z2" are then two complex variables, the condition that they have equal values can be written "z1.re = z2.re and z1.im = z2.im". And for some time I have felt that among the definition of the abstract type there must

be enough so as to be able to write a program (on the upper level!) that can determine whether two values of the abstract variable are equal. (As I have just done for the complex numbers.) I am no longer so sure of the reasonableness of the requirement in view of the fourth (potential)benefit. (In the example there is a way of establishing equality of the unaccessible ages, because "A older than B or B older than A" is equivalent with "A.age $\neq$ B.age".)

I am beginning to feel now, that "equality of values" is such a fundamental property that we should allow the algorithm establishing the equality of two values to be expressed on the lower level in terms of the values of the composing parts. The syntax suggested by Hoare would make this equality test an asymmetric algorithm, because he presented these lower level algorithms as within an inner block with one instance in the surrounding block. Also the operation "swap(a, b)", where a and b are variables of the same abstract type that should exchange their values is only very clumsily coded in his suggestion. In other words, Hoare's suggestion that the lower level algorithms are always executed subordinate to one instance of the abstract variable and will have explicit access to the components of that variable only, seems too restrictive. Up till now I do not have a better proposal.

* * *

Finally I draw attention to the fact that the introduction of fancy types may be expected to increase the number of partial functions and operations. With the current sequencing primitives the upper level programmer has to sequence his program explicitly in such a way that none of these functions or operations is invoked outside its domain. For robustness sake, however, the lower level implementation will start checking ... that the operation is not invoked outside its domain. First you give a complicated duty to the upper level programmer, next you check dynamically that the upper level programmer has not violated any of the rules! I mention, hust for the record, that my tentative notion of the "guarded command" is intended to present a better interface.

NUENEN - 16th November 1973 prof.dr.Edsger W.Dijkstra