to professor C.A.R.Hoare
Department of Computer Science
The Queen's University of Belfast
BELFAST BT7 1NN
Northern Ireland

Monday 31 August 1970

Dear Tony,

May I ask you a few things? My first question is very simple: I should like a new copy of "FIND", the paper you sent to the members of WG2.3, because I lost mine -someone borrowed mine- before I had paid all attention to it that it deserves.

My second question is more demanding: I should like to have your comments on the following thoughts. I ask this to you because on the subjects dealt with I consider you to be the world's expert. (Yet I hesitate as my question might be very demanding: I cannot [redacted] predict how much re-adjustment from you side is required to follow my considerations. In all probability I use the usual words -such as "value" and "reference" in a slightly different meaning!)

I associate with a variable the following things

1) it may have a name -"x" say- so that a program may refer to it

2) it will have a finite life-time, during which it will have a unique identity

3) it will be "of a type", i.e. associated with it will be a collection of possible values

4) at any moment during its life-time it will have one of the values of its type ("any moment" with respect to a sufficiently coarse grain of time, in which "changing its value" is an instantaneous operation)

5) given the identity of a variable, its value is uniquely defined; as many variables may have equal values, given its value, its identity is inggeneral not uniquely defined

6) if "x" is the name of a variable, in program textst "_ref_ x" will stand for its identity, "x" will stand for its value with two exceptions, viz.

a) when preceded by "_ref_"

b) when standing at the left of an assignment operator

7) besides the reference prefix we have the evaluation operator "_val_". Its function is given by

_val_ _ref_ x = _val_ (_ref_ x) = x .

Exception 6b) is introduced for compatibility reasons mainly. It allows us to write in the usual way

x:= x + 1

instead of the more cumbersome

_ref_ x:= x + 1 ;

if I wanted to do this with the assignment procedure "assign" I would have to write

assign(_ref_ x, x + 1).

from E.W.Dijkstra to C.A.R.Hoare, monday 31 August 1970 

If "y = ref x" then "val y = x", so I could write also

    val y:= x + 1        or        assign(y, x + 1)  .

The assignment operator does two things: it assigns a new variable to a variable (with an identity,i.e. for which we need its identity), it also destroys its old value. Exception 6b) tells us that our notation convention is to write down the indication of the value destroyed, the value replaced:

    "x:= x + 1" is read:

"the value of x is replaced by its value plus one". If the left hand side starts with val, the reference in question is found by omitting "val", otherwise by prefixing it with "ref". (At the bedinning I was very unhappy about this convention, after some time I found that it makes sense: you mention "old" and "new" value.)

If y is a value of type "reference" its type comprises the type of val y.

Given three real variables a, b and c, compare the two programs

1) printing the maximum value.

```
var max: real;
if a > b then max:= a else max:= b;
if c > max do max:= c;
print(max)
```

2) changing the sign of the maximum value

```
var max: ref real;
if a > b then max:= ref a else max:= ref b;
if c > val max do max:= ref c;
val max:= - val max
```

With the above two examples I have made my first point, viz. to find a clean interface expressing when evaluations have to take place, when I operate on values, when on identities. Any well-motivated suggestion about syntactic sugar (for the declaration for instance) is wellcome.

The next point deals with types and type control When declaring a variable, its type should be given either

    var prime: boolean

i.e. naming a type, or

    var prime: (true, false)

i.e. enumerating its possible values.

The latter arrangement suggest to write

```
type bintree:
        (tree1,
         tree2(number: real, left:bintree, right:bintree))
```

Here we introduce a type called "bintree" with two mutually exclusive values, either "tree1" ,which is used to denote an empty tree, or "tree2" which is a node comprising three variables called "number" (of type "real") "left" and "right" both of type "bintree".

Upon assignment of the value tree2, three suitable values should be supplied as actual parameters.

We can now write the boolean procedure "in" testing whether a given value is in a tree or not and adding it to the tree when it is not.

```
boolean procedure in(T: ref bintree, num: val real)
      case val T of
     (= tree1: {in:= false; val T:= tree2(num, tree1, tree1)},
      = tree2: {case number(val T) of
                (= num :{in:= true}.
                 > num :{in:= in(ref left(val T), num)}.
                 < num :{in:= in(ref right(val T), num)}}).
```

or, if you prefer a non-recursive procedure

```
boolean procedure in(T: ref bintree, num: val real)
begin var curtree ref bintree:= T;                   (this is an initializing decl.)
      [redacted]
      var found: boolean:= false;
      var diff: real;
      while val curtree = tree2 and non found do
      begin diff:= number(val curtree) - num;
            if diff = 0 then found:= true
                        else
              curtree:= if diff > 0 then ref left(val curtree)
                                    else ref right(val curtree)
      end;
      if non found do val curtree:= tree2(num, tree1, tree1);
      in:= found
end
```

To use the type and the procedure one can declare

```
var TREE: bintree:= tree1; in(ref TREE, 3.25)  .
```

A complication arises when we want to write a procedure removing the minimum element from a tree. In the heading we want to describe that initially it should be a non-empty tree; as a result of the call, however, it could become empty. I suggest as heading

```
real procedure minmove (T: ref tree2 → bintree)
```

telling that initially the actual parameter is restricted to the subtype "tree2" while eventually it may be of the general type bintree. A non-recursive version then is (after the above heading)

```
begin var curtree: ref bintree:= T;
      var leftree: ref bintree:= ref left(val T);
      while val lefttree = tree2 do
            begin curtree:= leftree; leftree:= ref left(val leftree) end;
      minmove:= number(val curtree); val curtree:= right(val curtree)
end
```

(Note: the heading of "in"could be:

```
boolean procedure in(T: ref bintree → tree2, num: val real)      )
```

In all this I am aiming at an interface such that a translator can do all type checking. I have a strong feeling that if this cannot be done, one has the wrong concept of type! To ease this analysis I am willing to add syntactic sugar (in the form of the type-controlled case clause of the type-controlled while clause.) I hope that a translator can check that the call of "minmove (<u>ref</u> TREE) will only occur with TREE = tree2.

I know that you have paid much attention to such questions in an earlier stage of your life: it is only now that I am beginning to become ripe for such questions and would value your comments very much. Yours are fine shoulders to stand upon!

Thank you for your handwritten letter with the sketch of a proof for the transposition algorithm. I am glad that you enjoyed the algorithm. Some fifteen years ago I was introduced to Christopher Strachey by Aad van Wijngaarden at some conference. Aad and I entered a restaurant and Christopher was sitting alone at a table, Aad asked whether we could join him and we were introduced. Christopher posed the problem and within five minutes -this was the time when Iwas still a bright boy!- I produced the solution. Some months ago the problem was posed to me again, I recognized the problem but had forgotten my solution: this second time it took me ten minutes to reconstruct it. This just shows what age does to you! My grapevine tells me, that you have shown the algorithm both to Brian Randell and Niklaus Wirth. Small world we live in!

Returning for a last time to the main subject of this letter: I am looking for a program representation from which it is at any moment obvious which values are defined (and how) and which are not. The two mutually exclusive forms of the value of a variable of type bintree are suggested by the analogy of the recursive procedure which must contain a <u>condition</u> call upon itself. The node which is either empty or "a set of fields" is the spatial analogue of "<u>if</u> B <u>do</u> S" which, when executed takes one of two mutually exculsive forms, either empty or S.

Shall I have the privilege of your answer to my questions?

Yours ever

Edsger

prof.dr.Edsger W.Dijkstra