Sequencing and the discriminated union.

The purpose of this note is to record an observation on a connection between the availability of sequencing primitives on the one hand and the need for the discriminated union on the other.

Our starting point is a rather abstract inner block that captures a structure of which I have encountered several examples. The variable z represents the global environment symbolically, the variable x is a local variable, and the predicates H(x) and K(x) , used in the annotations, are complementary, i.e. H(x) = non K(x) . The BHH , BHK , etc. represent boolean expressions such that (BHH(x,z) or BHK(x,z)) ⇒ H(x) , etc.

```
begin var x: Xtype; x:= some function(z);
   do BHH(x,z) → {H(x)} z:= ZHH(x,z); x:= XHH(x,z) {H(x)}
    [] BHK(x,z) → {H(x)} z:= ZHK(x,z); x:= XHK(x,z) {K(x)}
    [] BKH(x,z) → {K(x)} z:= ZKH(x,z); x:= XKH(x,z) {H(x)}
    [] BKK(x,z) → {K(x)} z:= ZKK(x,z); x:= XKK(x,z) {K(x)}
   od
end
```

Suppose now that we have to code this inner block in the absence of the required Xtype , but in the presence of two types, Htype and Ktype , such that there is a natural one-to-one correspondence between the values in Htype and those x of Xtype satisfying H(x) , and, similarly, there is a natural one-to-one correspondence between the values in Ktype and those x of Xtype satisfying K(x) . The classical solution consists of replacing the above variable x by a triple, i.e. one boolean, one variable of Htype and one variable of Ktype . Reusing the same identifiers in analogous functions, we get:

```
begin var Hholds: boolean; var h: Htype; var k: Ktype; Hholds:= Q(z);
   if Hholds → h:= some function(z) [] non Hholds → k:= other function(z) fi;
   do Hholds cand BHH(h,z) → z:= ZHH(h,z); h:= XHH(h,z)
    [] Hholds cand BHK(h,z) → z:= ZHK(h,z); k:= XHK(h,z); Hholds:= false
    [] non Hholds cand BKH(k,z) → z:= ZKH(k,z); h:= XKH(k,z); Hholds:= true
    [] non Hholds cand BKK(k,z) → z:= ZKK(k,z); k:= XKK(k,z)
   od
end
```

This second program is ugly for a variety of reasons:

1) The fact that at any moment in time of the values of h and k only one matters is not syntactically expressed.

2) Without the introduction of "fake initializations", the assignments to h and k cannot be separated in the text into initializations versus modifications. (This complaint is closely related to the first one.)

3) The value of Hholds requires explicit manipulation, despite the fact that that value is almost a function of the place in the text.

4) The cand's are really necessary, because the BHH , BHK etc. may now be partial functions. (Note that we may not write

```
do Hholds → if BHH(h,z) → ...
            [] BHK(h,z) → ...
            fi
 [] non Hholds → if BKH(k,z) → ...
                 [] BKK(k,z) → ...
                 fi
od
```

because, instead of to proper termination, this would lead to abortion.)

These complaints are largely overcome when we use --very much in the style suggested by Eric C.Hehner of the University of Toronto-- what we might call "semi-recursion". The main text for our program part then becomes

```
if Q(z) → processHtype(some function(z))
 [] non Q(z) → processKtype(other function(z))
fi
```

with the two local refinements, which --under the assumption of value-parameters in the style of ALGOL 60-- can be written:

```
processHtype(value h: Htype):
begin do BHH(h,z) → z:= ZHH(h,z); h:= XHH(h,z) od;
      if BHK(h,z) → z:= ZHK(h,z); processKtype(XHK(h,z))
       [] non BHK(h,z) → skip
      fi
end
```

and for "processKtype" similarly. Following Hehner we can abolish the do...od completely --and, in passing, retain the potential nondeterminacy-- by refining as follows:

```
processHtype(value h: Htype):
begin  if BHH(h,z) → z:= ZHH(h,z); processHtype(XHH(h,z))
        [] BHK(h,z) → z:= ZHK(h,z); processKtype(XHK(h,z))
        [] non (BHH(h,z) or BHK(h,z)) → skip
       fi
end
```

and for "processKtype" similarly.

This form of recursive refinement is at most "semi-recursion", for what an ALGOL programmer would intuitively interpret as calls on recursive procedures are here all so-called "last calls": for their implementation no stack is required and --because the term "continuation" has already a technical meaning in denotational semantics-- we could call them "completions". If the "completion" is a recognized syntactic concept, none of the four complaints against our second program applies to our semi-recursive programs!

*   *   *

The above can be read as a plea for semi-recursion as a sequencing device. Its strength, however, remains to be ascertained. Semi-recursion provided a nice alternative for our second program, i.e. a second way of avoiding a discriminated union --a way of type formation about which I have mixed feelings-- , but we should not forget that in this example we replaced only one variable of such a type. The complete moral of this observation --if there is one-- has still to be written; for the time being I must be content with having discovered a connection --between sequencing and types-- of which I had been unaware before.

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
BURROUGHS Research Fellow