Computation versus program.

I will use two simple examples to illustrate a very basic experience. Two computations that produce the same output are equivalent in that sense and a priori not in any other.

In the relation between program and computation we observe the program spread out in text space and the computation spread out in time. For any given combination of program and computation the so-called sequencing describes how progress of the computation (as time "progresses") is mapped on progress through the program (as text "progresses").

What is emerging are ways to compare programs; one wants to do so in order to compare the corresponding computations. The basic experience is that it is impossible (or fruitless, or terribly hard or unattractive or what you wish) to compare computations by comparing the corresponding programs when on the level of comparison the sequencings through the two programs differ.

In a little bit more detail: when we can parse two computations as a sequence of actions and we can map the two sequences of actions on each other, we can compare them by comparing the program texts, provided that the program texts can be equally parsed in instructions, each of them corresponding to an action.

Let me give two examples, an abstract one and a concrete one. The abstract example is the following one. Excluding side effects of boolean inspection and B2 constant

```
"while B1 do  if B2 then S1
                    else S2"
```

is equivalent with

```
"if B2 then while B1 do S1
       else while B1 do S2"    .
```

The first construction is primarily one in which sequencing is controlled by a repetition clause, the second construction is primarily one in which sequencing is controlled by an alternative clause. I can establish the equivalence of the output of the computations but I cannot regard them as equivalent in any other useful sense.

The concrete example is to construct a program generating SEQUENCES non-empty sequences of 0's, 1's and 2's without non-empty, element-wise equal adjoining subsequences, generating these sequences in alphabetic order until a sequence of length 100 (i.e. of 100 digits) has been generated. (The start of the list of sequences to be generated is:

0
01
010
0102
01020
010201
0102010
0102012)

The programmer may make use of the knowledge that a sequence of length 100 satisfying the conditions actually exists. Each solution (apart from the first one) is an extension (with one digit) of a solution and the algorithm is therefore a straightforward backtracking one.

We are looking for the "good" sequences, we assume a primitive available investigating whether a trial sequence is good. If it is good, the trial sequence is printed and extended with a zero to give the next trial sequence; if the trial sequence is no good we perform on it the operation "increase" to get the next trial sequence, i.e. final digits = 2 are removed and then the last digit remaining is increased by 1. (The existence of a solution of length 100 and our stopping there will see to it that removal of final digits = 2 will never give rise to an empty sequence.)

Version 1a uses the fact that a single zero is the first true solution.

```
Version 1a:
    "Set trial sequence to single zero (and length to 1);
     while length < 101 do
        begin  if good then
           begin print trial sequence; extend trial sequence with zero end
                       else
           increase trial sequence
        end"
```

Version 1b regards the empty sequence as a virtual solution, not to be printed:

```
Version 1b:
    "Set trial sequence empty (and length to 0);
     while length < 100 do
        begin extend trial sequence with zero;
              while no good do increase trial sequence;
              print trial sequence
       end"
```

One marked difference is in the statement to be repeated. XXXXXXXXX In Version 1a (conditional) printing of a solution preceeds the generation of a next trial, in Version 1b the printing is at the end of the repeated statement. This difference explains the difference in initialization and repetition test. But this is a minor difference as Version 1c shows:

```
Version 1c:
     "Set trial sequence to single zero (and length to 1);
     while length < 101 do
        begin while no good do increase trial sequence;
              print trial sequence;
              extend trial sequence with zero
        end"
```

The tremendous difference is, that in version 1a the two repetitions are merged into one, while version 1b can be regarded as a detailing of version 0b:

```
     "Set current sequence to virtual solution (and length to 0);
     while length < 100 do
        begin transform current sequence to next solution;
              print current sequence
        end"
```

Versions 1a and 1b are fairly incomparable. That was my basic experience.

On reflection I shall ask attention for a third example as it presents a border case. Given two arrays X[1:N] and Y[1:N]and a boolean "equal", make a program that gives to "equal" the meaning "the two given arrays are element-wise equal". Empty arrays are regarded as equal.

One cannot compare the two arrays at a single stroke, one has to do so element-wise; we introduce the integer "j" and give to the variable "equal" the following meaning "among the first j pairs of elements no difference has been found" and arrive at the following program part

Version 1:
```
j:= 0; equal:= true;
while j < N do
    begin j:= j + 1; equal:= equal and (X[j] = Y[j]) end
```

This does the job: the initial situation is in accordance with j = 0, the statement under the repetition clause implements the induction step from j to j + 1 (no difference so far and no new difference) and by the time that j = N we have the desired value.

Inspecting the assignment

"equal:= equal and ....."

we can conclude that once "equal = false"holds, this relation will be permanent and therefore further execution of the repetition clause makes no sense. (Mind you, we are only interested in the final value of "equal".) This observation gives rise to the following program section

Version 2:
```
j:= 0; equal:= true;
while j < N and equal do
    begin j:= j + 1; equal:= equal and (X[j] = Y[j]) end
```

But now we have made the program in such a way that the repeated statement will only start execution with "equal = true" and as a result "equal and" can be omitted:

Version 3:
```
j:= 0; equal:= true;
while j < N and equal do
    begin j:= j + 1; equal:= (X[j] = Y[i]) end
```

and that, presumably, will be our final version.

The above is a form of "program patching" that I abhor. For instance, the conclusion that led to Version 2 was derived from reading Version 1; Version 2 is fairly ridiculous anyway, it only occurred as a stepping stone between the other two versions. The question is: how are Version 1 and Version 3 related to each other? The sequencing is different, yet sufficiently similar that I can map them on each other.

We have N+1 functions EQUAL[j] for $0 \leq j \leq N$, defined upon the arrays and given by

```
EQUAL[0] = true
EQUAL[j] = EQUAL[j-1] and (X[j] = Y[j])
```

and in terms of these functions it is requested to perform the assignment

equal:= EQUAL[N] .

The common ancestor of Versions 1 and 3 would be something like

```
j:= 0; equal:= EQUAL[0];
while "perhaps equal ≠ EQUAL[N]" do
    begin j:= j + 1; "equal:= EQUAL[j]"end
```

Now this is tricky and not too well formulated. Each time the inspection is done, the relation "equal = EQUAL[j]" will hold because the common ancestor is made that way. At each inspection either "equal = EQUAL[N]" or not; I have included the word "perhaps" and have put the conditions within quotes, just to be on the safe side, in order to indicate whether we dare to guarantee that the equality holds. If we refuse to give this guarantee, well then "perhaps" the inequality holds.

Another way of saying why I have put the inspection within quotes is that what I have given what meaning I shall attach to the truth and falsity of the boolean expression, without stating what expression it is.

Our choice for the inspection depends on our lazyness, on the amount of mathematical analysis we wish to spend on the definition of the functions EQUAL. We can be lazy and say just: well, at the moment of inspection I know that

$$\text{equal} = \text{EQUAL}\,[j]$$

and I refuse to conclude that

$$\text{equal} = \text{EQUAL}\,[N]$$

before

$$j = N$$

holds. This leads to Version 1.

We can apply some analysis to the recurrence relation and conclude that for any j

$$\text{EQUAL}[j] = \underline{\text{false}} \text{ implies } \text{EQUAL}[i] = \underline{\text{false}} \text{ for all } i \geq j \ .$$

The class of situations under which we are now willing to guarantee the equality "equal = EQUAL[N]" is then widened to "j = N or equal = false" and this leads to version 3.

Now the really tricky thing is the following. We can regard the inspection "perhaps equal = EQUAL[N]" as an open primitive to be chosen later on; but the choice we make defines the set of circumstances under which the statement to be repeated has to be executed. In version 1, the only thing we can do is to follow the recurrence relation litterally. In version 3 the computation of EQUAL[j] is restricted to the case EQUAL[j-1] = true, so what is demanded of the other quoted action

"equal:= EQUAL[j]"

depends on the choice of the inspection. In version 3 it can be implemented by

"equal:= (X[j] = Y[j])" or "if X[j] ≠ Y[j] then equal:= false" ,

using the here known fact that initially "equal = true" will hold.

My common ancestor is an awkward parent!