A note on "tail invariants"

The purpose of this note, which is mainly for the record, is to describe a phenomenon which is well-known (and has been named) in Eindhoven, but deserves to be known more widely. In addition, we might be able to offer some explanation of the phenomenon.

The archetypical problem - archetypical since it was dealt with in the Ph.D. Thesis of James King - is in this context "exponentiation", specified by

```
|[ con X,Y: int {$X>0 \wedge Y\geq 0$}
 ; var z: int
 ; exponentiation {$z=X^Y$}
 ]|        .
```

Our first solution, exp0, does not use a tail invariant.

```
exp0: |[ var y: int
       ; y,z := 0,1 {P0: $0\leq y\leq Y \wedge z\geq 1 \wedge z=X^y$}
       ; do y ≠ Y → y,z := y+1, X·z od
         {P0 ∧ y = Y, hence}
       ]| {$z=X^Y$}
```

Our second solution, exp1, does use what is known as a tail invariant.

```
exp1: |[ var y : int
      ; y, z := Y, 1 {P1: $0 \le y \le Y \wedge z \ge 1 \wedge z \cdot X^y = X^Y$}
      ; do y ≠ 0 → y, z := y-1, X·z od
       {P1 ∧ y=0, hence}
     ]| {$z = X^Y$}
```

In the repetion we used in both programs the property of exponention – the "step"

$$X^{y+1} = X \cdot X^y \quad ;$$

in both programs we used the property of the exponentiation – the "base" –

$$X^0 = 1 \quad ,$$

but the difference is where it was used. In exp0, the base was used to justify the initialization (of y) – while the implication of the postcondition by $P \wedge \neg B$ requires only Leibniz's Principle. In exp1, the justification of the initialization requires Leibniz's Principle but no properties of exponentiation: the base is needed for the final implication of the postcondition.

* * *

The general experience is that the tail invariant is to be preferred: its use gives rise to the nicer derivations or the nicer programs or both. The explanation has probably many facets, one of them possibly being that the mathematical setting of exp1 is slightly richer than that of exp0, in which —in contrast to exp1— no use is made of the fact that 1 is the neutral element of the multiplication. More important is that the tail invariant gives greater flexibility.

In both programs —with X^y in the invariant— z's final value X^Y is primarily treated as a function of Y. That z has its final value X^Y has to be concluded from "$P \wedge \neg B$".

In exp0, Y does not occur in the invariant P0, therefore Y has to occur in the guard. In exp1, it is the other way round: Y occurs in invariant P1 but not in the guard. And here you see a major difference: in the invariant we can allow Y to occur via a nonprimitive subexpression like X^Y, but doing so in the guard would be begging the question.

(We reject

```
exp2: |[z:= 1  { P2: $1 \leq z \leq X^Y$ }
        ; do $z \neq X^Y \rightarrow z := z+1$ od
        ]| { $z = X^Y$ }
```

for obvious reasons.)

The greater flexibility of the tail invariant immediately manifests itself when we consider the value to be computed as a function of both X and Y :

```
exp3: |[ var x, y: int;  x, y, z := X, Y, 1
            {P3: $1 \le x \wedge 0 \le y \le Y \wedge 1 \le z \wedge X^Y = z \cdot x^y$}
        ; do y ≠ 0 →
                if odd.y → y, z := y-1, x·z
                ▯ even.y → x, y := x·x, y/2
                fi {P3}
          od {P3 ∧ y = 0 , hence}
        ]| {z = $X^Y$}     ,
```

in which we could exploit $x^{2\cdot n} = (x^2)^n$ to turn a linear algorithm into a logarithmic one. For the "base" it exploits in the final justification of the postcondition that $x^0 = 1$ (regardless of how high a power of X x is).

It is the last observation that pinpoints the problem with exp0, when we would like to introduce the variable x there too. In order to establish —analogously to P0—

$$z = x^y \quad ,$$

we could, so to speak, initialize

$$x, y, z := ?, 0, 1 \quad ,$$

where any positive "?" would do. It is hard to exploit that freedom in a constructive fashion, and for the time being it seems a wise precaution to become very suspicious when confronted with an initialization in which the value of one of the variables is irrelevant.

It all strongly suggests to shape our calculations as computations of recursively defined functions and to justify that initializations establish invariants by an appeal to Leibniz's Principle and not to properties of the functions to be computed. We note that (pure) functional programming languages go a long way towards syntactic enforcement of this discipline.

Austin, 1 October 1993

prof. dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA