RRIVE

ANOTHER DIFFERENCE BETWEEN RECURSIVE REFINEMENT AND REPETITION

by

Maarten M. Fokkinga
Twente University of Technology
P.O. Box 217, 7500 AE Enschede
The Netherlands

May 1978

Abstract

It is shown that - when derived with the same reasoning - [Hehner 76]'s recursively refined iterations deliver more acceptable results than [Dijkstra 76]'s repetitive constructs do.

Keywords and phrases: recursive refinement, repetition, nondeterminacy, weakest precondition, strongest postcondition.

Contents

Introduction

Nondeterminacy seems to be an advantage for the process of program composition, because of elegance. The more elegant a program is constructed, the easier it can be understood and proved correct or incorrect. Let us try to be more specific.

Internal nondeterminacy allows and forces guarded commands in one same set to be constructed independently of each other. Thus the program is purged from irrelevant and accidental ordering of some construction steps, and symmetric problems may be treated symmetrically. The conventional if then else and while do hardly allow, let alone encourage, this.

External nondeterminacy allows the delivery of several final states, all satisfying the required postcondition. Thus equally acceptable results may be delivered equally well. It might even be preferred that any final state satisfying the postcondition can indeed be reached from some initial state. Let us call this property: purity (; however, terminology might be improved). Purity might be desirable as a methodological principle, because it prevents users from exploiting unspecified properties of the program. (Purity seems to be complementary to robustness: any initial state not satisfying the weakest precondition leads to abortion of program execution.) To say it in other words, a program P is called pure w.r.t. a postcondition S if the strongest postcondition of P w.r.t. wp(P,S) is S again!

Note. We could have defined a program P to be pure w.r.t. S iff for any initial state any state satisfying S can be reached. But then the program P, which given a glovar x and required to establish true, should "set x to any value": this requires unbounded nondeterminacy and makes the concept useless. (End of note.)

Example 1. The following program is taken from [Dijkstra 76]p. 53. The relation to be established is

$0 \leq k < n$ and $\underline{A}\ i : 0 \leq i < n : f(i) \leq f(k)$,

that is, k should be some index for the greatest value in f (, n and f are fixed). Note that any of the acceptable values can be delivered in k. The program is pure.

```
k,j := 0,1;
do j ≠ n → if f(j) ≤ f(k) → j := j+1
         □ f(k) ≤ f(j) → k,j := j,j+1
         fi
od.
```

(End of example 1.)

Example 2. For fixed n and f, a program is requested for the establishment of

$0 \leq k < n$ and $f(k-1 \bmod n) \leq f(k) \geq f(k+1 \bmod n)$.

A pure program is

```
"set k to any value ≥ 0 and < n";
do f(k-1 mod n) > f(k) → k := k+1 mod n
□ f(k) < f(k+1 mod n) → k := k+1 mod n
od.
```

The purity crucially depends on the purity of the first statement w.r.t. $0 \leq k < n$. We will return to this problem later. (End of example 2.)

The difference

The difference between recursive refinement and do od, we like to point out, is that the former allows for more purity than the latter does. More precisely, there are situations in which recursive refinement delivers more acceptable results than a do od construct -- derived with the same (patterns of) reasoning -- can do. We like to consider this an advantage of recursive refinement over do od.

The proof is easy. First, in a recursively refined iteration, the alternatives may independently specify whether to recur or not. So nondeterministically the iteration may terminate, having established the required postcondition, or continue, maintaining the apparently established required postcondition. (The independent specification whether to recur or not also enables the multi-level exits; see [Hehner 76].)

Second, the continuation alternatives will of course have been derived by the same reasoning as in a do od iteration. The termination alternatives have guards guaranteeing the required post-iteration condition; although these guards do not explicitly occur in a do od construct, the necessary reasoning has to be made anyway.

The point is that mostly the termination conditions do exclude the (weakest) preconditions of the commands which make measurable progress, but sometimes they do not! (End of proof.)

The proof immediately seems to show a disadvantage. Whenever an iteration may terminate or continue, termination is to be preferred for reasons of efficiency. However, we consider it an aspect of a subsequent engineering phase to trade gain in efficiency versus loss of purity -- if so desired --.

A trivial example reads as follows. For fixed m and n, satisfying $0 \leq m \leq n$, it is required to establish $m \leq k' \leq n$ by the following assignments to k: k := 0 and k := k+1. Cfr. example 2.

The following programs may be derived formally, by almost the same reasoning for each.

P1: k := 0; do $\neg m \leq k \rightarrow$ k := k+1 od.

P2: k := 0; do $k \neq n \rightarrow$ k := k+1 od.

P3: k := 0; "x", where

"x" : if $m \leq k \rightarrow$ skip $\square$ $k \neq n \rightarrow$ k := k+1; "x" fi.

A nontrivial example: SPLIT as used in QUICKSORT

Given an integer array glovar a, integers glocon m,n satisfying $a\cdot lob \leq m \leq n \leq a\cdot hib$, and integer virvar l,r. It is requested to establish

S: $m-1 \leq l < n$ and $a(m..r-1) \leq a(l+1..n)$ and $m < r \leq n+1$ and $l < r$,

using a:swap(,) as the only assignment operation to a.

Thus a(m..n) is split effectively ($l < n$ and $m < r$) into a leftpart a(m..l) of small elements, a rightpart a(r..n) of large elements and a middle part a(l+1..r-1) of equal elements in value in between the small and large ones.

My attempts to give a fully satisfactory derivation of the algorithm SPLIT has led me to discover the concept of purity, and that purity is another difference between recursive refinement and do od. I admit however, that up to now I have found no other nontrivial example which may be used to illustrate purity. This may explain why that concept has not been discovered earlier, but it may also indicate that the concept is not a very practical one; I leave it open for discussion.

In [Fokkinga 78] we formally derive the following three programs, each one with almost exactly the same reasoning. The invariant relation has been obtained by deleting the term l < r from S. With one minor exception*), each guard precisely implies -- in its context -- the required wp(-,-) and wdec(-,-). It should further be noted that the programs below are just a stepping stone. In subsequent engineering phases the efficiency may be improved by the introduction of redundant variables high and low, by strengthening some guards, by making then some implicit determinacy explicit (so that unnecessary evaluation of guards is avoided) and so on. Eventually, any of the well known algorithms may be obtained [Hoare 61, Wirth 76, Van Emden 70].

The initialisation is the same for all three:

```
if a(m) ≤ a(n) → skip □ a(m) ≥ a(n) → a :swap(m,n) fi;
r vir int, l vir int := m+1, n-1;
```

The iterations, all establishing S, read as follows.

```
P1 : do r ≤ l →
        if a(r) ≤ a(l+1..n) → r := r+1
        □  a(m..r-1) ≤ a(l) → l := l-1
        □  a(m..r-1) ≤ a(r) ≥ a(l) ≤ a(l+1..n) → a :swap(r,l); r,l := r+1,l-1
        fi
     od.
P2 : do r ≠ n+1 cand a(r) ≤ a(l+1..n) → r := r+1
     □  m-1 ≠ l cand a(m..r-1) ≤ a(l) → l := l-1
     □  r ≤ l cand a(m..r-1) ≤ a(r) ≥ a(l) ≤ a(l+1..n) →
                    a :swap(r,l); r,l := r+1,l-1
     od.
P3 : "XXX":
     if l < r → skip
     □  r ≠ n+1 cand a(r) ≤ a(l+1 .. n) → r := r+1; "XXX"
     □  m-1 ≠ l cand a(m..r-1) ≤ a(l) → l := l-1; "XXX"
     □  r ≤ l cand a(m..r-1) ≤ a(r) ≥ a(l) ≤ a(l+1..n) →
                    a :swap(r,l); r,l := r+1,l-1; "XXX"
     fi .
```

---

*) We have omitted the term ... or m-1 ≠ l < r ≠ n+1 cand a(l) = a(r) in the swap-guard in P2 and P3.

Indeed, P1 establishes a minimal difference between r and l (i.e. l = r-1 or l+1 = r or l+1 = r-1), P2 a maximal difference (i.e. m-1 = l cor a(l) < a(l+1), similar for r), and P3 is the only one which can deliver any difference between these extremes. In fact P3 is pure (; any array can be delivered as well -- from some suitable initial state!--).

References

Dijkstra, E.W., 1976: A discipline of programming, Prentice Hall Inc., Engelwood Cliffs, N.J.

Van Emden, M., 1970: Increasing the efficiency of Quicksort, C.A.C.M. 13 (1970) 563-567, 693-694.

Fokkinga, M.M., 1978: in preparation

Hehner, E.C.R., 1976: A contribution to the programming calculus, University of Toronto, Revised Januari 1978.

Hoare, C.A.R., 1961: Algorithm 63 Partition, Algorithm 64 Quicksort, C.A.C.M. 4 (1961) 321.

Wirth, N., 1976: Algorithms + Data Structures = Programs, Prentice Hall, Inc., Englewood Cliffs, N.J.

Note added in proof. The programs P1 and P3 in section "The difference" and P3 in section "A nontrivial example", can be made more robust by replacing:

in P1: $\neg m \leq k$ by $m \neq k$,

in P3: $\neq$ by $<$.

But these changes require some additional reasoning and could therefore be considered to belong to a subsequent "robustness increasing phase". (End of note.)