Rotating a linear array

In the following, X and Y will stand for string constants, A, B, C, D and the linear array $f(i: 0 \leq i < N)$ will stand for string variables, and concatenation will be denoted by just juxtaposition; #.X denotes the length of string X. We are given the functional specification

```
|[ N, K: int {0<K<N}
; |[ f(i: 0≤i<N): array
    { f = XY ∧ #.X = K}
  ; Rot
    { f = YX }
  ]|
]|
```

What do we know about concatenation of strings? Well, that it is associative - i.e $(AB)C = A(BC)$ - and that it has the empty string, which we denote by $\emptyset$, as identity element, - i.e. $A\emptyset = A \wedge \emptyset A = A$ -.

The knowledge of the identity element is of importance because it enables us to insert into the postcondition a constant, which we can then replace by a variable, B say. Without destruction of the symmetry between left and right we can rewrite the postcondition as $f = Y \emptyset X$ and head for the invariant $f = Y B X$. But that won't do on account of the precondition. So what about

two more variables A and C ? With $f = A\,B\,C$ we can characterize the postcondition by $(A, B, C) = (Y, \phi, X)$ and the precondition by $(A, B, C) = (X, \phi, Y)$. But this won't do the job either, because it would mean that the variable that replaced the constant in the postcondition has to be initialized with its final value!

That one empty string inserted in the postcondition did not give us enough freedom, but we are getting close because, again without destruction of the symmetry between left and right, two empty strings can be inserted in two different ways, "in between" and "around". That is, if we rewrite the postcondition as $f = Y\,\phi\,\phi\,X$, we could rewrite the precondition as $f = \phi\,X\,Y\,\phi$. With

P0 : $f = A\,B\,C\,D$

we would have to transform

$$(A, B, C, D) = (\phi, X, Y, \phi) \qquad (0)$$

into $(A, B, C, D) = (Y, \phi, \phi, X)$ (1)

Remark 0 Alternatively, we could have transformed

$$(A, B, C, D) = (X, \phi, \phi, Y)$$

into $(A, B, C, D) = (\phi, Y, X, \phi)$.

Our analysis so far gives no reason for strong preference. In our choice, we have upon completion $B\,C = \phi$, the alternative would have $A\,D = \phi$. In contrast to the string $B\,C$, the string $A\,D$

is not a substring of f, the array the manipulations on which will eventually make up the algorithm. This is a reason for preferring the transition from (0) to (1). (End of Remark 0.)

Having chosen our initial and our final state, what would now be our invariant? A reasonable suggestion is

P_1: $\quad A\,C = Y \;\wedge\; B\,D = X$.

Remark 1 There is the alternative

$$C\,A = Y \;\wedge\; D\,B = X$$

that we shall look at later. We have chosen to investigate P_1 first because in it the concatenations occur in the order in which the strings occur in f - see P_0 -. (End of Remark 1.)

Let us now look for simple multiple assignment statements that reduce #.(C B) under invariance of P_1. The most effective one reduces #.(C B) to zero at one fell swoop:

S_0: $\quad A, C, B, D := (A\,C), \phi, \phi, (B\,D)$

Let f(=front) and r(=rear) be used to denote the partitioning of a string such that (f.Z r.Z) = Z for any string Z. Then P_1 is maintained by the assignment statements

S_1: $\quad A, C := (A\ f.C), r.C$

S_2: $\quad B, D := f.B, (r.B\ D)$

The above statements, designed so as to maintain P_1 have to maintain

P_0: $\quad f = A\ B\ C\ D$

by manipulating f and we now try, for reasons of efficiency, to confine those manipulations to swaps of disjoint array segments of equal length. To this purpose we investigate for our statements the weakest precondition with respect to P_0.

(S_0) $\quad wp(S_0, P_0)$
$= \{$Axiom of Assignment$\}$
$\quad f = (A\ C)\ \phi\ \phi\ (B\ D)$
$= \{$Properties of Concatenation$\}$
$\quad f = A\ C\ B\ D$

which, thanks to the initial validity of P_0 can be achieved by swapping B and C. As we had decided to restrict swaps to array segments of equal length, for $\#.B = \#.C$ we are done.

(S_1) $\quad wp(S_1, P_0)$
$= \{$Assignment and Concatenation$\}$
$\quad f = A\ f.C\ B\ r.C\ D$

which, thanks to the initial validity of P_0, i.e.

$$f = A\ B\ f.C\ r.C\ D$$

can be achieved by swapping B and $f.C$. The constraint $\#.B = \#.(f.C)$ can be met if $\#.B < \#C$.

(S_2) $\quad wp(S_2, P_0)$
$= \{$Assignment and Concatenation$\}$
$\quad f = A\ f.B\ C\ r.B\ D$

which, thanks to the initial validity of P0, i.e.

$$f = A\ f.B\ r.B\ C\ D$$

can be achieved by swapping r.B and C. The constraint of $\#.(r.B) = \#.C$ can be met if $\#.B > \#.C$

In terms of A, B, C, and D this leads to

```
  A,B,C,D := ∅, X, Y, ∅   {#.B ≥ 1 ∧ #.C ≥ 1}
; do #.B < #.C → S1
   [] #.B > #.C → S2
  od
; S0
```

With the representational convention

$A = f(i: 0 \le i < k-b)$
$B = f(i: k-b \le i < k)$
$C = f(i: k \le i < k+c)$
$D = f(i: k+c \le i < N)$

we thus find for Rot

```
[[ b,c,k: int ; b,c,k := K, N-K, K
; do b<c → f: segswap.(k-b).k.b ; k,c := k+b, c-b
   [] b>c → f: segswap.(k-c).k.c ; k,b := k-b, b-c
  od
; f: segswap.(k-b).k.c
]]
```

in which f: segswap.x.y.z swaps the array segments $f(i: x \le i < x+z)$ and $f(i: y \le i < y+z)$.

It looks as if we can now safely discard the suggestion of Remark 1. The analogues to S0, S1, and S2 are straightforward enough, but the analysis of the requirement to maintain P0 leads to manipulations of f, wilder than what the whole program is supposed to achieve.

A closer analysis of our solution reveals a curiosity. We have used that our postcondition $f = \Upsilon X$ is implied by $P0 \wedge P1 \wedge BC = \phi$ but it is in fact implied by the weaker $P0 \wedge P1 \wedge (B = \phi \vee C = \phi)$, the computation can already terminate as soon as one of the two is empty. An unexploitable feature as B and C turn empty simultaneously. The observation raises the question whether we can replace P1 by a weaker invariant such that the last conjunct $BC = \phi$ is not stronger than necessary. A weaker P2 would be

$$P2: \quad A\,C\,B\,D = \Upsilon X \quad ,$$

which, yet would be strong enough since

$$\begin{aligned}
& P0 \wedge P2 \wedge BC = \phi \\
= & \ \{\text{definitions and concatenation}\} \\
& f = A\,B\,C\,D \wedge A\,C\,B\,D = \Upsilon X \wedge (B = \phi \wedge C = \phi) \\
\Rightarrow & \ \{\text{Leibniz}\} \\
& f = A\,\phi\,\phi\,D \wedge A\,\phi\,\phi\,D = \Upsilon X \\
\Rightarrow & \ \{\text{Leibniz}\} \\
& f = \Upsilon X \quad .
\end{aligned}$$

(Note: $\vee$ instead of $\wedge$ would still suffice!)

S0 maintains P2 as well as it maintains P1, and I propose to keep it for dealing with the case that $\#.B = \#.C$. But gives the weaker P2 freedom to replace S1 and S2? Let us try.

P_2 is maintained by

S3: $A, C, B := (A\ C), f.B, r.B$

S4: $C, B, D := f.C, r.C, (B\ D)$,

which decrease $\#.(B\ C)$ by $\#.c$ and $\#.b$ respectively. We investigate as before their weakest preconditions with respect to P_0

(S3) $wp(S3, P_0)$
$= \{$Assignment and Concatenation$\}$
$f = A\ C\ r.B\ f.B\ D$

which, thanks to the initial validity of P_0, i.e.

$f = A\ f.B\ r.B\ C\ D$

can be obtained by swapping $f.B$ and C ; S3 is applicable with $\#.B > \#.C$

(S4) $wp(S3, P_0)$
$= \{$Assignment and Concatenation$\}$
$f = A\ r.C\ f.C\ B\ D$

which, thanks to the initial validity of P_0, i.e.

$f = A\ B\ f.C\ r.C\ D$

can be obtained by swapping B and $r.C$; S4 is applicable with $\#.B < \#.C$.

Coding with the same representational convention as before we discover that k's inialization is the only assignment to k ; hence that variable is not needed and, therefore, another solution for Rot is

```
|[ b,c: int ; b,c := K, N-K
; do b>c → f: segswap.(K-b).K.c ; b:= b-c
  [] b<c → f: segswap.(K-b).(K+c-b).b ; c:= c-b
 od
; f: segswap.(k-b).k.c
]|
```

Note We only bothered to define f: segswap.x.y.z for natural z. Had we extended for negative z a swap of the array segments f(i: x+z ≤ i < x) and f(i: y+z ≤ i < y), then the last guarded command would have been

$$b<c \rightarrow f: \text{segswap}.K.(K+c).(-b);\ c:= c-b\ .$$

It is probably proper to define f: segswap.x.y.z as a swap of

$$f(i: (x \ \underline{\min}\ x+z) \leq i < (x \ \underline{\max}\ x+z)) \quad \text{and}$$

$$f(i: (y \ \underline{\min}\ y+z) \leq i < (y \ \underline{\max}\ y+z)).$$

(End of Note.)

We still have the suggestion of Remark 0. We could explore the invariant $P_0 \wedge P_3$ with

$$P_3: \quad AC = X \ \wedge \ BD = Y$$

and now our target is to get $A = \emptyset \wedge D = \emptyset$. The situation is discouragingly different from that with the invariant $P_0 \wedge P_1$, where the post-condition was already implied if either $B = \emptyset$ or $C = \emptyset$. Further exploration shows that we do run

into trouble and that the suggestion of Remark 0 is not to be recommended. This is not surprising. Swapping two disjoint array segments is its own inverse, and Remark 0 in essence suggests to invert one of our earlier solutions (in the sense of playing them backwards). The earlier solutions perform on b and c Euclid's algorithm for the gcd(N,K) and inversion of Euclid's algorithm is not an attractive proposal. Hence we discard the suggestion of Remark 0.

* * *

Concatenation is a simple dyadic operator on strings, nicely associative and all that. There is also a simple monadic operator on strings, nice because it is its own inverse, viz. the reverse of a string: if #.Z ≤ 1, rev.Z = Z, and furthermore rev.(XY) = rev.Y rev.X

or

rev.(rev.X rev.Y) = Y X

which yields a very different solution for Rot

```
|[ i,j: int
; i,j := 0, K-1 ; do i<j → f:swap.i.j ; i,j := i+1, j-1 od
; i,j := K, N-1 ; do i<j → f:swap.i.j ; i,j := i+1, j-1 od
; i,j := 0, N-1 ; do i<j → f:swap.i.j ; i,j := i+1, j-1 od
]|
```

where f:swap.i.j interchanges the values of f.i and f.j.

* * *

For Rot we could have given the -equivalent- specification

```
|[ N, K: int {0<K<N}
; |[ f(i: 0≤i<N): array {(A i: 0≤i<N: f.i = F.i)}
  ; Rot
    {(A i: 0≤i<N: f.i = F.((i+K) mod N))}
  ]|
]|        .
```

It is possible to develop all the above algorithms without the introduction of string variables, i.e. carrying out the whole derivation in terms of universally quantified equalities between elements of f and of F, as in the above functional specification. When you try it, you will discover that the exercise is an obfuscating pain in the neck. Had we started from the above functional specification, the invention of X and Y would probably have been our first duty.

(The above specification has often lead to yet another solution for Rot, in which the required permutation is written as the product of cyclic permutations. In order to determine those, gcd(N, K) has to be computed.)

Austin, 28 September 1985

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