Programmes Corrects par Construction
(ii) Exemples

1 Segment AB maximal

Parmi les segments du tableau f dont la première valeur est A, dont la dernière valeur est B, trouver l'un de ceux dont la somme des éléments est maximale. Il faut d'abord formaliser cette spécification :

```
R: r = (↑i,j : 0≤i<j<N ∧ f.i=A ∧ f.j=B : S.i.j)

avec S.i.j = (Σk : i≤k≤j : f.k)

  var A,B : int
; var f(0..N-1) : int {1≤N}
; var r : int
r: R
```

Nous proposons de construire un invariant en transformant la constante N en une variable :

```
P0: 1≤n≤N
P1: r = (↑i,j : 0≤i<j<n ∧ f.i=A ∧ f.j=B : S.i.j)
```

Ce qui amène à l'esquisse de programme suivante :

```
  n:=1 ; r:=-∞
; {inv P0 ∧ P1} {bf N-n}
  do n≠N ->
    r: S
  ; {?P1[n\n+1]}
    n:=n+1
  od
  {R}
```

Pour que bf évolue, l'instruction n:=n+1 semble forcée. Par définition de l'affectation, pour que l'invariant soit maintenu, il faut que la précondition de l'instruction n:=n+1 soit au moins aussi forte que l'assertion P1[n\n+1]. Ainsi, nous cherchons à établir un programme S qui réalise cette assertion. Ce programme S doit mettre à jour la valeur de r afin que P1[n\n+1] soit vérifiée. Ainsi, nous cherchons à réaliser P1[n\n+1] pour calculer le programme S qui mettra à jour la variable r.

```
Re P1[n\n+1]

  (↑i,j : 0≤i<j<n+1 ∧ f.i=A ∧ f.j=B : S.i.j)
= { split pour j=n, correct car n≥1 par P0,
    P1 }
  r ↑ (↑i : 0≤i<n ∧ f.i=A ∧ f.n=B : S.i.n)
= { soit f.n=B, soit f.n≠B, correct car 0≤n<N,
    le max sur un domaine vide vaut −∞}
  if f.n≠B -> r
    |f.n=B -> r ↑ (↑i : 0≤i<n ∧ f.i=A : S.i.n)
  fi
```

A ce point, nous aimerions que la valeur : (↑i : 0≤i<n ∧ f.i=A : S.i.n) ait été calculée avant la mise à jour de la variable r. Nous introduisons une variable s qui devra permettre de connaitre cette valeur. Puisque nous voulons utiliser la valeur de s dans le calcul de la mise à jour de r, il faut que s ait été mise à jour avant r. Comme nous introduisons s, qui sera mise à jour au sein de la boucle, pour assurer la correction du programme il faut renforcer l'invariant pour qu'il inclue la description de la valeur de s :

```
P2: s=(↑i : 0≤i<n−1 ∧ f.i=A : S.i.(n−1))

  n:=1 ; r:=−∞ ; s:=−∞
; {inv P0 ∧ P1 ∧ P2} {bf N−n}
  do n≠N ->
    s: T
  ; {?P2[n\n+1]}{P0}{P1}
    if f.n≠B -> skip
      |f.n=B -> r:= r↑s
    fi
  ; {P1[n\n+1]}
    n:=n+1
  od
  {R}
```

Remarquons que lorsque nous vient l'idée d'introduire la variable s, c'est-à-dire au cours de la réalisation de P1[n\n+1], nous rencontrons une formule que nous interprétons comme s sous sa forme mise-à-jour. D'où la différence entre cette formule et P2.

Il nous reste maintenant à réaliser P2[n\n+1] pour construire le programme T qui permettra la mise-à-jour de la variable s.

```
Re P2[n\n+1]

   (↑i : 0≤i<n ∧ f.i=A : S.i.n)
= { S.i.n = f.n + S.i.(n−1),
    + distribue sur ↑ }
   f.n + (↑i : 0≤i<n ∧ f.i=A : S.i.(n−1))
= { split sur i=n−1, correct car 1≤n d'après P0,
    P2 }
   f.n + (s ↑ (↑i : i=n−1 ∧ f.i=A : S.i.(n−1)))
= { S.(n−1).(n−1) = f.(n−1),
    max sur un domaine vide vaut −∞,
    alternative }
   if f.(n−1)≠A -> f.n + s
     |f.(n−1)=A -> f.n + (s ↑ f.(n−1))
   fi
```

D'où le programme :

```
   var A,B : int
; var f(0..N−1) : int {1≤N}
; var r,s : int

; n:=1 ; r:=−∞ ; s:=−∞
   do n≠N ->
   if f.(n−1)≠A -> s:= f.n + s
     |f.(n−1)=A -> s:= f.n + (s↑f.(n−1))
   fi
   ; if f.n≠B -> skip
       |f.n=B -> r:= r↑s
     fi
   ; n:=n+1
   od
```

2 Plus long segment nul

```
  var N: int {N≥0}
; var f(0..N−1): int
; var r: int
r: R

R: r = (↑p,q : 0≤p≤q≤N ∧ Z.p.q : q−p)

Z.p.q = (∀i : p≤i<q : f.i=0)
```

Nous appliquons la stratégie consistant à transformer une constante en une variable pour construire un invariant. Ce qui nous amène à l'esquisse de programme suivante :

```
P0: r = (↑p,q : 0≤p≤q≤n ∧ Z.p.q : q−p)
P1: 0≤n≤N
R:
  n,r := 0,0
  {P0 ∧ P1}
; do n≠N ->
      r: S
      {?P0[n\n+1]}
   ; n:=n+1
  od
```

Pour dériver le programme S qui mettra à jour la variable r, nous cherchons à réaliser P0[n\n+1].

```
Re P0[n\n+1]

  (↑p,q : 0≤p≤q≤n+1 ∧ Z.p.q : q−p)
= { split sur q=n+1, correct car n≠N par la clause de garde et
    n≤N par P1,
    P0 }
  r ↑ (↑p : 0≤p≤n+1 ∧ Z.p.(n+1) : n+1−p)
= { + distribue sur ↑ }
  r ↑ (n+1 + (↑p : 0≤p≤n+1 ∧ Z.p.(n+1) : −p))
= { dualité min/max }
  r ↑ (n+1 − (↓p : 0≤p≤n+1 ∧ Z.p.(n+1) : p))
```

Nous renforçons l'invariant :

```
P2: s = (↓p : 0≤p≤n ∧ Z.p.n : p)
```

Nous obtenons l'esquisse de programme suivante :

```
  n,r,s := 0,0,0
  {inv P0 ∧ P1 ∧ P2, bf N–n}
; do n≠N –>
     s: T
     {?P2[n\n+1]}
   ; r:= r↑(n+1–s)
   ; n:=n+1
  od {R}
```

Pour calculer le programme assurant la mise-à-jour de la variable s, nous réalisons la vérité de P2[n\n+1].

```
Re P2[n\n+1]

  (↓p : 0≤p≤n+1 ∧ Z.p.(n+1) : p)
= { split sur p=n+1, correct car n≠N par la clause de garde et
    n≤N par P1,
    Z.(n+1).(n+1)=true }
  (n+1)↓(↓p : 0≤p≤n ∧ Z.p.(n+1) : p)
= { soit f.n≠0 et le domaine du min est vide
                donc sa valeur est ∞,
    soit f.n=0 et la valeur du min nous est donnée par P2 }
  (n+1)↓(if f.n=0 –> s
          |f.n≠0 –> ∞
         fi)
```

D'où le programme :

```
  var n,r,s: int
; var N: int {N≥0}
; var f(0..N–1): int
; n,r,s := 0,0,0
; do n≠N –>
     if f.n=0 –> skip
       |f.n≠0 –> s:=n+1
     fi
   ; r:= r↑(n+1–s)
   ; n:=n+1
  od
```

3 Division entière

Nous reprenons l'exemple de la division entière introduit dans la première partie du cours :

```
R: 0≤r ∧ r<y ∧ q*y+r = x

   var r,q,x,y : int
   {Q: x≥0 ∧ y>0}
; q,r: R
```

La post-condition étant une conjonction, nous adoptons la stratégie de retirer un conjoint pour construire un invariant :

```
inv    P: 0≤r ∧ q*y+r = x
garde B: r≥y
```

D'où l'esquisse de programme :

```
   {Q: 0≤x ∧ 0<y}
   q,r:= 0,x
   {inv P: 0≤r ∧ q*y+r = x, bf r}
; do r≥y -> S od
   {P ∧ r<y, d'où: R}
```

Pour assurer l'évolution de la bound-function (bf: r). Le corps de la boucle doit avoir la forme suivante : r,q:= r−k, E

```
   {Q: 0≤x ∧ 0<y}
   q,r:= 0,x
   {inv P: 0≤r ∧ q*y+r = x, bf r}
; do r≥y -> r,q:= r−k, E od
   {P ∧ r<y, d'où: R}
```

Le corps de la boucle doit maintenir l'invariant :

```
{P ∧ r≥y} r,q := r−k,E {P}
```

```
    wp.(r,q := r−k,E).P
= {déf. P et affectation}
    r≥k ∧ E*y + r−k = x
= {P}
    r≥k ∧ E*y + r−k = q*y + r
= {arithmétique}
    r≥k ∧ E = q + k/y
```

Nous avons :

```
0<k ∧ k≤r ∧ E=q+k/y
```

Dans la première partie du cours, nous remarquions que k devait être un multiple de y pour ne pas avoir à utiliser la division, et nous choisissions k=1∗y. Ici, notre stratégie devient :

— (a) Choisir k aussi grand que possible.
— (b) Ne pas utiliser de division.

Cette stratégie nous guide pour élucider la forme de E :

```
  0<k ∧ k≤r ∧ E=q+k/y
= { k=d∗y selon (b) }
  0<d∗y ∧ d∗y≤r ∧ E=q+d
= { 0<y, donc d>0≡d≥1 }
  1≤d ∧ d∗y≤r ∧ E=q+d
= { hyp. 1≤d ∧ d∗y≤r }
  E=q+d
```

Ce qui nous amène à l'esquisse de corps de boucle suivante :

```
{P ∧ r≥y}
|[ var d: int
 ; S0: d: 1≤d ∧ d∗y≤r
 ; r,q:= r−d∗y, q+d
]|
{P}
```

La notation |[...]| est utilisée pour introduire des variables dans un contexte local. S0 étant le programme qui va calculer la valeur de d, laissons lui le soin de calculer d*y :

```
{P ∧ r≥y}
|[ var d,dd: int
 ; S0: d,dd: 1≤d ∧ d∗y≤r ∧ dd=d∗y
 ; r,q:= r−dd, q+d
]|
{P}
```

La post-condition de S0 est composée de trois conjoints :

```
R0: 1≤d
R1: d∗y ≤ r
R2: dd = d∗y
```

Selon la stratégie (a), nous cherchons une grande valeur pour d. La plus grande serait la plus petite vérifiant : r<(d+1)∗y. Cette valeur serait calculable par une recherche linéaire avec incrément de taille 1... Nou aimerions aller plus vite...!

Nous proposons de restreindre les valeurs de d aux puissances de 2 :

```
R3: d est une puissance de 2
```

Par la stratégie (a), nous voulons que d soit la plus grande puissance de 2 possible :

```
R4: r<2*d*y
```

Ainsi, la post-condition de S0 devient :

```
R0 ∧ R1 ∧ R2 ∧ R3 ∧ R4
```

Pour construire l'invariant, nous cherchons à retirer l'un des conjoints. R0 et R3, en tant que propriétés essentielles de d ne sont pas des bons candidats. Non plus R2 qui est le seul conjoint à mentionner la variable dd. Puisque nous avons l'intention de faire croître d de puissances de 2 en puissances de 2, il nous semble pertinent de retirer le conjoint R4 de l'invariant. Nous obtenons pour S0 l'esquisse de programme suivante :

```
inv P0: R0 ∧ R1 ∧ R2 ∧ R3

  d,dd:= 1,y
  {inv P0, bf d}
; do r≥2*d*y -> S00 od
```

L'invariant est maintenu avec :

```
S00: d,dd:=2*d, 2*dd
```

D'où le programme final dont, pour des raisons peut-être seulement esthétiques, nous avons retiré toute multiplication :

```
  var q,r: int
; q,r:= 0,x
; do r≥y ->
    |[  var d,dd: int
      ; d,dd:= 1,y
      ; do r≥dd+dd -> d,dd:= d+d, dd+dd od
      ; r,q:= r−dd,q+d
    ]|
  od
```

4 Dutch national Flag (DNF)

Soit un tableau f d'objets marqués r, w ou b. Le trier pour que les r soient en premières positions, suivis des w, suivis des b.

```
c.l.h.x ≙ (∀ i : l≤i<h : f.i=x)

  var N: int {N≥0}
; var f(0..N−1): rwb
  R: (∃ wb,bb : 0≤wb≤bb≤N : c.0.wb.r ∧ c.wb.bb.w ∧ c.bb.N.b)
```

Nous adoptons la stratégie de transformer une constante en une variable pour construire l'invariant. Il s'agit d'introduire un segment du tableau dont la couleur (r, w ou b) des éléments est inconnue. Pour ne pas trop introduire d'asymétrie, nous n'introduisons pas ce segment à une des deux extrémités du tableau :

```
.-----------------------.
|  r  |  w  |  u  |  b  |
.|-----|-----|-----|----.|
 v     v     v     v     v
 0     wb    ub    bb    N
```

```
Pr: c.0.wb.r  Pw: c.wb.ub.w  Pb: c.bb.N.b

inv P: Pr ∧ Pw ∧ Pb ∧ 0≤wb≤ub≤bb≤ N
```

Nous obtenons l'esquisse de programme suivante pour la réalisation de r :

```
P ∧ (ub=bb) ⇒ R

  var wb,ub,bb: int
; wb,ub,bb := 0,0,N
{inv P, bf bb−ub}
; do ub≠bb ->
    if f.ub = r -> Sr
     |f.ub = w -> Sw
     |f.ub = b -> Sb
    fi
  od
```

Sw semble simple à réaliser :

```
Sw: ub := ub+1
```

Pour Sr, nous avons l'idée suivante (qui utilise une affectation multiple, i.e. toutes les affectations se font en parallèle) :

```
Sr: f.wb, f.ub, wb, ub := f.ub, f.wb, wb+1, ub+1
```

Avec l'incrément de ub, cette réalisation de Sr fait trivialement évoluer la bound-function. Vérifions qu'elle maintienne l'invariant P.

Par la clause de garde de la répétition, nous avons : ub≠bb donc Pb est trivialement maintenu.

Vérifions que Pw soit maintenu :

```
  wp.Sr.Pw
= { déf. de := }
  Pw[f.wb,f.ub,wb,ub \ f.ub,f.wb,wb+1,ub+1]
= { substitution }
  (∀i : wb+1≤i<ub+1 : f.i=w)[f.ub \ f.wb]
= { split sur i=ub }
  (f.ub=w)[f.ub \ f.wb] ∧ (∀i : wb+1≤i<ub : f.i=w)
= { substition et Pw }
  true
```

De même nous vérifierions que Pr soit maintenu, puis nous ferions un raisonnement similaire pour Sb, pour arriver au programme suivant :

```
  var N: int {N≥0}
; var f(0..N−1): rwb
; var wb,ub,bb: int
; wb,ub,bb:=0,0,N
; do bb≠ub ->
    if f.ub=r -> f.wb,f.ub,wb,ub := f.ub,f.wb,wb+1,ub+1
     |f.ub=w -> ub := ub+1
     |f.ub=b -> f.ub,f.(bb−1),bb := f.(bb−1), f.ub, bb−1
    fi
  od
```

5 Quicksort

Soit un tableau A (mettons d'entiers pour fixer les idées) qu'il faut trier. L'idée est d'appliquer itérativement le DNF.

```
  var N: int {N≥0}
; var A(0..N−1): int
```

Soit z=A.j avec 0≤j<N. Nous pouvons appliquer le DNF avec :

```
r ≜ A.i < z , w ≜ A.i=z , b ≜ A.i>z
```

Ce qui nous permet d'établir :

```
(∀i : 0≤i<wb: A.i<z) ∧
(∀i : wb≤i<bb: A.i=z) ∧
(∀i : bb≤i<N: A.i>z)
```

Ensuite, nous voulons recommencer le processus sur les deux parties non triées. A un instant donné de ce processus itératif, nous aurons un ensemble de segments disjoints de A qui ne seront pas triés.

Ainsi, posons V un ensemble d'intervalles disjoints sur $[0..N-1]$.

Nous avons l'idée de l'invariant suivant :

```
inv P: A est trié ≡ (∀ v : V : A est trié sur v)
```

On obtient l'esquisse de programme suivante :

```
  var N: int {N≥0}
; var A(0..N−1): int
; var V: set of intervals of int
; var α: interval of int
; var j: int
; var z: int

; V := {[0..N−1]}

; do V≠∅ −>
    "choisir α ∈ V"
  ; if len.α ≤ 1 −> V := V\{α}
     |len.α>1 −>
        "choisir j ∈ α"
      ; z := A.j
      ; DNF sur α
      { α = βγδ ∧ (∀ i : β : A.i < z)
                ∧ (∀ i : γ : A.i = z)
                ∧ (∀ i : δ : A.i > z) }
      ; V := V \ {α} ∪ {β} ∪ {δ}
    fi
  od
```

Montrons que le programme termine. Nous proposons de prendre comme bound-function la taille de l'union des éléments des éléments de V. Cette fonction décroît lors du passage par la seconde branche du test conditionnel. Elle décroît ou reste stable lors du passage par la première branche du test conditionnel. Elle est stable pour toujours si et seulement si V ne contient que des ensembles vides, sa valeur est alors de 0. Ainsi, elle atteint un jour 0. Quand elle atteint 0, le programme emprunte un nombre fini de fois la première branche de l'instruction conditionnelle.

Nous proposons de représenter V avec deux tableaux d'entiers :

```
  var M: int
; var x(0..M−1): int
; var y(0..M−1): int
; var k: int

{ V = {i : 0≤i<k : [x.i..y.i[} }
```

Quel espace allouer aux tableaux x et y ? Autrement dit, quelle est la taille maximale de V ? Nous minimisons la taille maximale de V si à chaque fois de "choisir $\alpha \in V$", nous choisissons un élément de taille minimale. Autrement dit, il faut toujours privilégier l'emprunt de la première branche du test conditionnel, seul lieu du programme où la taille de V diminue.

Par ailleurs, nous savons qu'un des segments β ou δ est d'une taille inférieure à la moitié de la taille de α. Nous notons G.n une borne supérieure sur la taille maximale atteinte par V si (i) son élément initial est de taille n et que (ii) l'élément α retiré de V est choisi de taille minimale :

$G.1 = 1$
$G.n = 1 + G.(n \text{ div } 2)$

Résolution de la récurrence :
$G.n = 1 + \lceil log_2.n \rceil$

Puisque diviser un plus petit intervalle de V génère un nouveau plus petit intervalle, nous pouvons facilement conserver V trié et identifier les bornes (viz. p et q) du plus petit segment actuel :

$V = \{i : 0 \leq i < k : [x.i\,..\,y.i[\} \cup \{[p\,..\,q[\}$
$(\forall\ i : 0 < i < k : (y.i - x.i) \leq (y.(i-1) - x.(i-1)))$
$(\forall\ i : 0 \leq i < k : (q-p) \leq (y.i - x.i))$

Ainsi, nous avons :

$P \wedge k=0 \wedge (q-p) \leq 1 \Rightarrow$ A est trié

Et la clause de garde de la boucle principale devient :

$(k \neq 0) \vee (q-p) > 1$

D'où le programme :

```
  var N: int {N≥0}
; var A(0..N−1): int
; var j: int
; var z: int
; var M: int
; var x(0..M−1): int
; var y(0..M−1): int
; var k,p,q: int
; var wb,ub,bb: int

; var n: int
; M,n := 0,1;
{inv n = 2^M, bf n}
; do n<N −> M,n := M+1,n∗2
; M := 1+M
{M = 1 + ⌈log_2.N⌉}

; k,p,q := 0,0,N

; do (k≠0) ∨ (q−p)>1 −>
    if (q−p)≤ 1 −>
         k := k−1
       ; p,q := x.k, y.k
      |(q−p)>1 −>
         { le mieux serait de calculer la médiane... un bon exercice! }
         z := A.((p+q) div 2)
         { DNF sur α }
       ; wb,ub,bb := p,p,q
       ; do ub≠bb −>
            if A.ub<z −> A.wb,A.ub,wb,ub := A.ub, A.ub, ub, wb
              |A.ub=z −> ub := ub + 1
              |A.ub>z −> A.ub,A.(bb−1),bb := A.(bb−1),A.ub,bb−1
            fi
         od
       ; if (wb−p)≤(q−bb) −> x.k,y.k := bb,q ; q := wb
            |(q−bb)≤(wb−p) −> x.k,y.k := p,wb ; p := bb
         fi
         k:=k+1
    fi
  od
```