Multiprogrammes Corrects par Construction

1 Correction locale et globale

Un multiprogramme est l'exécution concurrente de plusieurs programmes séquentiels qui peuvent partager des variables.

Une assertion (i.e., une formule de la logique des prédicats utilisée pour annoter un programme, et qui doit être vérifiée après que l'instruction précédent l'assertion ait été exécutée) doit être :

— **localement** correcte, et

— **globalement** correcte.

La **correction locale** considère le composant où apparait l'assertion isolé du reste du système (i.e., des autres composants).

Ainsi, une assertion P au tout début d'un composant, avant qu'aucune instruction n'ait été exécutée, pour être correcte doit être impliquée par l'initialisation du multi-programme. Sinon, P est la post-condition d'une instruction S de précondition Q, et il faut vérifier la correction du triplet de Hoare {Q}S{P} comme nous le ferions pour un programme séquentiel.

La **correction globale** d'une assertion {P} est établie en prouvant que toutes les instructions {Q}S des composants autres que celui où apparait {P} maintiennent {P} : {P ∧ Q} S {P}.

P est un **invariant système** s'il est impliqué par les assertions vérifiées à l'initialisation du multi-programme et s'il est maintenu par chaque instruction {Q}S de chaque composant : {P ∧ Q} S {P}.

2 Modification de la sémantique de l'alternative

Pour un multi-programme, la sémantique de l'alternative est modifiée par rapport à la définition que nous en avions donnée pour les programmes séquentiels. Il n'est plus nécessaire qu'au moins une des clauses de garde soit vérifiée. Si aucune clause de garde n'est vérifiée, le composant est bloqué et il sera éventuellement débloqué par les actions d'autres composants qui permettront à au moins une des clauses de garde d'être vérifiée. Ainsi, nous utiliserons la forme suivante de **"if bloquant"** comme mécanisme de synchronisation :

```
if B -> skip fi
```

Cette dernière forme est équivalente à :

```
do ¬B -> skip od
```

En termes de triplets de Hoare, nous avons :

```
{B⇒R} if B -> skip fi {R}
```

Nous parlerons maintenant de **précondition libérale la plus faible (wlp)**. wlp.S.R est la précondition la plus faible pour que, *si le programme* S *termine*, il établisse R.

Ainsi, nous pouvons maintenant rencontrer des répétitions sans condition de sortie. Nous noterons ∗[S] pour do ⊤ −> S od.

3 Un exemple simple

```
Pre: x=0
Comp.0: *[ x:=x+1 ]
Comp.1: *[ print.x ]
```

Le premier composant, viz. Comp.1, doit afficher la suite des nombres entiers naturels. Nous pouvons le spécifier en introduisant une nouvelle variable :

```
Pre: x=0 ∧ y=0
Comp.0: *[ x:=x+1 ]
Comp.1: *[ {? x=y, voir Note.0}
           print.x
         ;y:=y+1
         ]
```

Note.0 : x=y

Correction locale (L) :
Préfixer x=y par if x=y −> skip fi. C'est-à-dire que la seule façon pour Comp.1 d'établir x=y est d'attendre que Comp.0 l'ait établie.

Correction globale (G) :
Il faut s'assurer que l'instruction x:=x+1 de Comp.0 ne puisse pas falsifier x=y :

```
  x=y ⇒ wlp.(x:=x+1).(x=y)
= {déf. :=}
  x=y ⇒ x+1=y
= {arithmétique}
  x≠y
```

Il faut donc ajouter x≠y en pré-assertion de x:=x+1.

```
Pre: x=0 ∧ y=0

Comp.0: *[ {? x≠y, voir Note.1}
           x:=x+1
         ]

Comp.1: *[ if x=y -> skip fi
           {x=y}
         ;print.x
         ;y:=y+1
         ]
```

Note.1 :
(L) :
Préfixer l'assertion x≠y avec if x≠y −> skip fi.
(G) :
L'instruction {x=y}y:=y+1 de Comp.1 doit maintenir x≠y :

```
  x≠y ∧ x=y ⇒  wlp.(y:=y+1).(x≠y)
= {logique}
  ⊤
```

```
Pre: x=0 ∧ y=0

Comp.0: *[ if x≠y -> skip fi
           {x≠y}
           x:=x+1
         ]

Comp.1: *[ if x=y -> skip fi
           {x=y}
         ;print.x
         ;y:=y+1
         ]
```

4 Safe Sluice, un essai pour l'exclusion mutuelle de sections critiques

```
Pre: ⊤

Comp.p:
  *[ snc.p
   ;sc.p
   ]

Comp.q:
  *[ snc.q
   ;sc.q
   ]
```

Objectif : Comp.p et Comp.q ne doivent pas être dans leurs sections critiques (viz. sc.p et sc.q) en même temps.

Par ailleurs, nous posons qu'**une section critique termine toujours** (contrairement à une section non critique qui peut ne pas terminer).

Nous introduisons deux variables pour pouvoir modéliser cet objectif sous la forme du maintien d'un invariant système :

```
y.p ≡ Comp.p est dans sc.p
y.q ≡ Comp.q est dans sc.q
```

```
Pre: ¬y.p ∧ ¬y.q

Inv: ? R: ¬y.p ∨ ¬y.q

Comp.p:
  *[ snc.p
   ;y.p := ⊤
   ;sc.p
   ;y.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Pour que y.p := ⊤ ne puisse pas falsifier R, il faut lui ajouter la précondition ¬y.q qui ne peut être établie que par Comp.q. Comp.p doit donc attendre que cette assertion soit vérifiée :

```
Pre: ¬y.p ∧ ¬y.q

Inv: R: ¬y.p ∨ ¬y.q

Comp.p:
  *[ snc.p
   ;if ¬y.q -> y.p := ⊤ fi
   ;sc.p
   ;y.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Interblocage :
Nous en profitons pour introduire le concept d'interblocage. Il y a interblocage si tous les composants sont en même temps en attente dans un if bloquant. Pour montrer l'absence d'interblocage, il faut montrer que la conjonction des conditions d'attente est fausse.

Sur notre exemple, l'interblocage est impossible :

```
  y.p ∧ y.q
= {logique}
  ¬(¬y.p ∨ ¬y.q)
= {déf. de R}
  ¬R
= {inv R}
  faux
```

Progrès :
Nous introduisons maintenant le concept de progrès individuel des composants. Sous l'hypothèse d'un comportement équitable de l'ordonnanceur (viz. tout composant sera

toujours un jour ou l'autre élu par l'ordonnanceur), il y a progrès individuel des composants s'il n'existe pas de séquences d'exécutions pour lesquelles un composant serait pour toujours bloqué.

Sur notre exemple, le progrès n'est pas assuré car nous pouvons construire un ordonnancement tel qu'à chaque fois que Comp.p a la main il soit bloqué sur son *if-bloquant*, tandis que Comp.q exécute alternativement ses sections critiques et non-critiques sans être bloqué.

Atomicité :

Un fragment de programme est dit **atomique** si son exécution ne peut pas être interrompue par l'ordonnanceur. Nous considérerons le plus souvent comme atomique un fragment de code qui ne contient qu'**au plus une lecture ou une écriture sur une variable partagée**.

Sur notre exemple, le *if-bloquant* n'est pas atomique. Par exemple, pour Comp.p il contient une lecture sur la variable partagée y.q et une écriture sur la variable partagée y.p. Ainsi, sous cette définition de l'atomicité, le code proposé n'est pas correct : les deux composants peuvent exécuter leurs sections critiques en même temps...

Pour réduire la granularité de l'atomicité, il est souvent possible d'employer la stratégie : *renforcer une clause de garde d'un if-bloquant ne modifie pas la correction d'un multi-programme* (bien que cela puisse mettre en danger la présence de progrès et l'absence d'interblocage...).

Pour employer cette dernière stratégie, nous introduisons les variables x.p et x.q telles que :

```
  ¬x.q ⇒ ¬y.q
= {logique}
  y.q ⇒ x.q
```

...soit établi en précondition de if ¬y.q −> y.p := ⊤ fi.

Nous proposons que y.q ⇒ x.q soit un invariant système.

Pour éviter que y.q := ⊤ ne falsifie y.q ⇒ x.q, nous pourrions proposer :

```
if ¬y.p −> y.q, x.q := ⊤, ⊤ fi
```

Mais ce serait s'éloigner un peu plus d'une atomicité à la granularité suffisamment fine...

Ainsi, nous proposons :

```
Pre: ¬y.p ∧ ¬y.q ∧ ¬x.p ∧ ¬x.q

Inv: R: ¬y.p ∨ ¬y.q ,
     y.p ⇒ x.p ∧ y.q ⇒ x.q

Comp.p:
  *[ snc.p
   ;x.p := ⊤
   ;if ¬y.q -> y.p := ⊤ fi
   ;sc.p
   ;y.p := ⊥
   ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

La correction de cette solution est due à la **topologie du multiprogramme** : i.e., seul Comp.p (resp. Comp.q) peut modifier la variable x.p (resp. x.q).

Maintenant l'invariant établi, nous pouvons remplacer ¬y.q par ¬x.q dans la clause de garde de Comp.p. x ayant pris le rôle de y, cette dernière n'est plus que *figurante* et peut disparaître :

```
Pre: ¬x.p ∧ ¬x.q

Comp.p:
  *[ snc.p
   ;x.p := ⊤
   ;if ¬x.q -> skip fi
   ;sc.p
   ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Remarque : Avec la disparition des variables y, l'invariant ¬y.p ∨ ¬y.q n'est plus visible. Par ailleurs, nous n'avons pas : ¬x.p ∨ ¬x.q.

Enfin, si cette transformation maintient bien l'exclusion mutuelle des sections critiques, elle autorise cependant l'interblocage...

5 L'algorithme de Peterson

Pour le programme *Safe Sluice*, l'interblocage est possible car x.p ∧ x.q peut être vrai. Nous adoptons la stratégie **d'affaiblir la clause de garde du if-bloquant pour éviter l'interblocage**. Cet affaiblissement peut mettre en danger la correction du programme, i.e. ne plus permettre l'exclusion mutuelle des sections critiques.

```
Pre: ¬x.p ∧ ¬x.q

Comp.p:
  *[ snc.p
   ;x.p := ⊤
   ;if ¬x.q ∨ H.p.q -> skip fi
   ; {? R.p.q} sc.p
   ;x.p := ⊥
  ]

Comp.q: Comp.p[p\q]
```

Il faut donc faire en sorte que :
— **(i)** L'assertion R.p.q (resp. R.q.p) soit correcte
— **(ii)** $R.p.q \wedge R.q.p \Rightarrow \bot$
Pour que (ii) soit vérifiée, nous cherchons une assertion R.p.q aussi forte que possible. En considérant la structure de Comp.p, l'assertion R.p.q la plus forte serait :

```
R.p.q ≜ x.p ∧ (¬x.q ∨ H.p.q)
```

Seule l'instruction x.q := ⊤ pourrait mettre en danger la correction globale de cette assertion. Ainsi, il faut que :
— **(iii)** `x.q :=` ⊤ de `Comp.q` établisse `H.p.q`
Ainsi, en établissant (iii), (i) sera établi. Essayons d'établir (ii) :

```
  R.p.q ∧ R.q.p
= {déf. de R}
  x.p ∧ (¬x.q ∨ H.p.q) ∧ x.q ∧ (¬x.p ∨ H.q.p)
= {logique}
  x.p ∧ H.p.q ∧ x.q ∧ H.q.p
⇒ {On ne sait rien à propos des x}
  H.p.q ∧ H.q.p
```

Pour établir (ii), il faut donc établir H.p.q ∧ H.q.p ⇒ ⊥. Nous préférons cette forme à (ii) car R est une *assertion* dont la forme est fortement contrainte par la structure du programme, tandis que le choix de H semble plus libre.

H.p.q et H.q.p sont des formules inconnues qui font intervenir les variables p et q. Nous venons de découvrir que leur conjonction doit être plus forte que ⊥. Pour progresser dans la dérivation, nous cherchons à exprimer ⊥ en fonction des variables p et q afin d'avoir peut-être une idée sur la forme de H. Nous remarquons que p≠q. Ainsi nous avons :

```
H.p.q ∧ H.q.p ⇒ p=q
```

Pour résoudre cette équation en trouvant une forme convenable aux formules inconnues H, nous nous concentrons sur la seule relation connue apparaissant dans cette équation : p=q. En observant que l'égalité est transitive, nous proposons d'*introduire une variable v* telle que :

```
H.p.q ≜ v=q
```

Ainsi, (ii) est établie. Il reste à établir (iii) pour vérifier (i). Nous pouvons simplement remplacer x.q := ⊤ par x.q,v := ⊤,q. Ainsi, nous obtenons :

```
Pre: ¬x.p ∧ ¬x.q

Comp.p:
  *[ snc.p
   ;x.p,v := ⊤, p
   ;if ¬x.q ∨ v=q -> skip fi
   ;sc.p
   ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

L'affectation multiple ne peut pas être considérée atomique. Pour résoudre ce problème, nous adoptons la stratégie de renforcement des clauses de garde en introduisant les variables y telles que :

```
¬y.p ⇒ ¬x.p ∧ ¬y.q ⇒ ¬x.q
```

Nous obtenons :

```
Pre: ¬x.p ∧ ¬x.q ∧ ¬y.p ∧ ¬y.q

Comp.p:
  *[ snc.p
   ;y.p:=⊤
   ;x.p,v := ⊤, p
   ;if ¬x.q ∨ v=q -> skip fi
   ;sc.p
   ;x.p:= ⊥
   ;y.p:= ⊥
   ]

Comp.q: Comp.p[p\q]
```

Puis nous remplaçons la garde ¬x.q ∨ v=q par ¬y.q ∨ v=q, et les variables x deviennent seulement figurantes et peuvent être retirées du programme. Et nous avons dérivé l'algorithme de Peterson :

```
Pre: ¬y.p ∧ ¬y.q

Comp.p:
  *[ snc.p
     ;y.p := ⊤
     ;v := p
     ;if ¬y.q ∨ v=q -> skip fi
     ;sc.p
     ;y.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

On vérifie facilement que l'interblocage est impossible :

$$
\begin{array}{ll}
 & \neg(\neg y.q \vee v = q) \wedge \neg(\neg y.p \vee v = p) \\
= & \{\text{De Morgan}\} \\
 & y.q \wedge v \neq q \wedge y.p \wedge v \neq p \\
= & \{v \neq q \wedge v \neq p = \bot\} \\
 & \bot
\end{array}
$$

Montrons que le progrès est assuré : un composant ayant terminé l'exécution de sa section non critique entrera dans sa section critique après un nombre fini d'instructions exécutées par le reste du système. Pour ce faire, nous montrons que sa condition d'attente, si elle n'est pas vraie, sera rendue vraie par l'autre composant et le restera stablement.

Ainsi, considérons Comp.q bloqué sur sa condition de passage : ¬y.p ∨ v=p. Montrons qu'alors Comp.p converge vers un état pour lequel cette condition est stablement vraie. Nous observons que la condition d'attente de Comp.p termine car il y a absence d'interblocage. Par ailleurs, sc.p termine par hypothèse. Ainsi Comp.p se réduit à :

```
*[ {¬y.p} snc.p
   ;y.p := ⊤
   ;v   := p
   ;y.p := ⊥
 ]
Comp.q: Comp.p[p\q]
```

Ainsi, en un nombre fini d'instructions, Comp.p rendra v=p invariablement vrai — donc également la condition de passage de Comp.q —, ou il sera dans une exécution de snc.p qui ne termine pas et ¬y.p — et donc aussi la condition de passage de Comp.q — sera invariablement vraie.

6 Peterson dérivé du point de vue du progrès

Soient deux composants p et q tels que :

```
Comp.p:
*[snc.p ; sc.p]

Comp.q:
*[snc.q ; sc.q]
```

Nous cherchons à assurer :

— **EM** : (Exclusion Mutuelle) cs.p et cs.q ne s'exécutent jamais ensemble

— **PI** : (Progrès Individuel) un composant qui a terminé sa section non critique snc, entre dans sa section critique sc après un nombre fini d'étapes du multiprogramme. *Vice versa.*

Ces exigences pourront être assurées par un mécanisme de synchronisation à déterminer :

```
Comp.p:
*[ snc.p
  ;entree.p
  ;if α -> skip fi
  ;sc.p
  ;sortie.p
 ]

Comp.q:
*[ snc.q
  ;entree.q
  ;if β -> skip fi
  ;sc.q
  ;sortie.q
 ]
```

Il y a absence de progrès si Comp.p est bloqué sur if α -> skip fi tandis que le reste du système n'établira plus jamais α. Nous pensons à trois situations qui pourraient occasionner l'absence de progrès :

(a) : $dansif.p \wedge dansif.q \wedge \neg\alpha \wedge \neg\beta$
avec dansif.p $\triangleq$ Comp.p exécute ou va exécuter if α -> skip fi
Pour éviter cette situation d'interblocage, il faut assurer :

```
(i) dansif.p ∧ dansif.q ⇒ α ∨ β
```

(b) : Comp.q est dans une exécution de sa section non critique snc.q qui ne terminera pas, alors que α est faux.
Pour éviter cette situation d'absence de progrès individuel, il faut assurer :

```
(ii)  dansif.p ∧ danssnc.q ⇒ α
(iii) dansif.q ∧ danssnc.p ⇒ β
```

(c) : snc.q et if β -> skip fi terminent, et Comp.q passe toujours devant Comp.p. Autrement dit, α oscille entre vrai et faux, et Comp.p évalue toujours α lorsqu'il est faux.
Pour éviter cette situation d'absence de progrès individuel, il faut assurer qu'au bout d'un moment les oscillations de α ne sont plus possibles :

```
Comp.p:
*[ snc.p
  ;s,t := s+1,0
  ;if α -> skip fi
  ;sc.p
 ]

Comp.q:
*[ snc.q
  ;t,s := t+1,0
  ;if β -> skip fi
  ;sc.q
 ]

Avec :
(∃ N : N ≥ 0 : (dansif.p ∧ t > N) ⇒ α), et
(∃ M : M ≥ 0 : (dansif.q ∧ s > M) ⇒ β)
```

Nous remarquons que (a), (b) et (c) sont trivialement établies avec α et β vraies. Mais l'exigence **EM** demande des valeurs fortes pour α et β. Or les équations (ii) et (iii) donnent les valeurs les plus fortes possibles pour α et β.

Ainsi, sous l'hypothèse de (ii) et (iii), cherchons à prouver (i) :

```
  α ∨ β
⇐ {(ii) et (iii)}
  dansif.p ∧ (danssnc.q ∨ t>N) ∨ (dansif.q ∧ (danssnc.p ∨ s>M))
= {antécédent de (i)}
  danssnc.q ∨ t>N ∨ danssnc.p ∨ s>M
= {antécédent de (i) : dansif.p ⇒ ¬danssnc.p}
  t>N ∨ s>M
```

```
(iv) t>N ∨ s>M
```

Sous hypothèse de (ii) et (iii), (iv) doit être rendue invariablement vraie pour assurer (i). Or, nous avons déjà les invariants :

```
  P0: 0≤s ∧ 0≤t
  P1: s>0 ≢ t>0
Donc :
      s>0 ∨ t>0
```

Ainsi, avec M,N = 0,0 : (ii)∧(iii) ⇒ (i). Nous avons donc déterminé que la valeur la plus forte possible pour α est :

```
   α ≡ dansif.p ∧ (danssnc.q ∨ t>0)
= { par définition, dansif.p est vrai en précondition de
      if α -> skip fi }
   α ≡ danssnc.q ∨ t>0

De même :
   β ≡ danssnc.p ∨ s>0
```

Nous introduisons les variables x.p et x.q pour modéliser les propositions danssnc.p et danssnc.q :

```
x.p ≡ danssnc.p,
x.q ≡ danssnc.q
```

Par ailleurs, remarquons que s et t ont une fonction binaire (viz. supérieure ou non à zéro). De plus, par P1 elles ne peuvent pas être positives en même temps. Donc, nous pouvons traduire les deux variables s et t en une unique variable binaire v :

```
v=p ≡ s>0,
v=q ≡ t>0
```

Nous avons dérivé le programme suivant :

```
Pre: (v=p ∨ v=q) ∧ (x.p ∧ x.q)

Comp.p:
*[ snc.p
  ;x.p := ⊥
  ;v := p
  ;if x.q ∨ v=q -> skip fi
  ;sc.p
  ;x.p := ⊤
 ]

Comp.q: par symétrie
```

Il s'agit de l'algorithme de Peterson. Ainsi, cet algorithme est non seulement correct (i.e., il assure l'exclusion mutuelle des sections critiques, voir précédente dérivation), mais il assure également le progrès.

7 Recherche linéaire parallèle

Nous considérons une structure de données qu'il faut parcourir linéairement pour trouver un objet dont on sait qu'il est un élément de cette structure. Notons f.i une valeur booléenne indiquant si l'élément cherché se trouve à l'indice i de la structure de données :

```
   (∃ i : domaine des indices de la structure de données considérée : f.i)
= {on considère une structure de données indexable sur les entiers}
   (∃ i : int : f.i)
```

Nous voulons mener cette recherche linéaire au moyen de deux composants d'un multiprogramme. Un composant explorera les entiers positifs, l'autre explorera les entiers négatifs :

```
Pre:   x=0 ∧ y=0
Inv:   0≤x ∧ y≤0
Post:  f.x ∨ f.y

Comp.0:
do  ... -> x:=x+1 od  {Q0}

Comp.1:
do  ... -> y:=y-1 od  {Q1}

Q0 ∧ Q1 ⇒ f.x ∨ f.y
```

En choisissant Q0 $\triangleq$ f.x $\vee$ f.y, pour assurer sa correction locale, il faudrait que la clause de garde de la boucle soit : $\neg$(f.x $\vee$ f.y) ce qui demanderait une granularité trop grossière pour l'atomicité.

Pour réduire la granularité de l'atomicité, nous essayons de renforcer Q0 et Q1 :

```
Q0 ≜ b,
Q1 ≜ b,
Inv P: b ⇒ f.x ∨ f.y
```

Pour la correction locale de Q0 (resp. Q1), la clause de garde de la boucle doit être : $\neg$b :

```
Pre:   x=0 ∧ y=0 ∧ ¬b
Inv:   0≤x ∧ y≤0
  P:   b ⇒ f.x ∨ f.y

Comp.0:
do ¬b -> x:=x+1 od  {b}

Comp.1:
do ¬b -> y:=y-1 od  {b}
```

Ce programme ne termine pas car $\neg$b est invariant. D'où :

```
Pre:   x=0 ∧ y=0 ∧ ¬b
Inv:   0≤x ∧ y≤0
  P:   b ⇒ f.x ∨ f.y

Comp.0:
do ¬b -> if B -> b:=⊤
           |C -> x:=x+1
         fi
od {b}

Comp.1:  par symétrie
```

Etant donné ce squelette de programme, nous avançons dans la dérivation en cherchant à réaliser l'invariant P.

Re P : b ⇒ f.x ∨ f.y
Init : ¬b vrai à l'initialisation établit l'invariant.
L : b:=⊤

```
  wlp.(b:=⊤).(b ⇒ f.x ∨ f.y)
= {déf. :=}
  f.x ∨ f.y
```

Pour avoir un programme qui termine, nous aimerions avoir un B aussi faible que possible. Mais, pour avoir une atomicité au grain suffisamment fin, et car la dépendance en f.y semble contraire à l'idée même du programme, nous proposons :

```
B ≙ f.x
```

Par souci de terminaison, nous aimerions un C aussi fort que possible. Cependant, un C plus fort que ¬f.x mettrait en danger le progrès puisqu'il serait possible qu'aucune des clauses de garde ne soit vérifiée sans qu'il soit possible à l'autre composant de rendre un jour une des clauses de garde vraies. Nous choisissons donc :

```
C ≙ ¬f.x
```

L : {¬f.x}x:=x+1

```
  wlp.(x:=x+1).(b ⇒ f.x ∨ f.y)
= {déf. :=}
  b ⇒ f.(x+1) ∨ f.y
```

Nous cherchons donc à prouver :

```
  P ∧ ¬f.x ⇒ b ⇒ f.(x+1) ∨ f.y
= {déf. P}
  b ⇒ f.y   ⇒ b ⇒ f.(x+1) ∨ f.y
= {logique}
  vrai
```

Finalement, nous obtenons le programme :

```
Pre:   x=0 ∧ y=0 ∧ ¬b
Inv:   0≤x ∧ y≤0
       ∧ (b ⇒ f.x ∨ f.y)
Post:  f.x ∨ f.y

Comp.0:
do ¬b -> if  f.x -> b:=⊤
            |¬f.x -> x:=x+1
          fi
od {b}

Comp.1:  par symétrie
```

Par ailleurs, nous avons :
(∀i:0≤i<x:¬f.i) ∧ (∀i:y<i≤0:¬f.i)
Or, (∃i::f.i)
Donc, soit x est bornée par en dessus, soit y est bornée par en dessous. Ainsi, un jour ou l'autre, b:=⊤ est exécuté et b reste vraie. Donc, le programme termine.