

Réponse

Le plus intéressant à faire est l'exercice 2, puisque l'algorithme à démontrer est faux. On en a pour une grosse heure de calcul à le montrer, mais en général, le résultat tombe comme un coup de tonnerre. Ceux qui se moquaient de la pénibilité de la preuve formelle "pour un code aussi simple que ça" sont bien étonnés quand on montre qu'il est faux. **Il ne faut pas leur dire à l'avance : ne spoilez pas le TD !**

Fin réponse

Dans ce TD, nous allons démontrer que les algorithmes étudiés sont corrects en calculant la plus faible précondition nécessaire (*weakest precondition*) pour que l'algorithme donne bien dans la post-condition souhaitée (le résultat est celui attendu). Cette méthode est l'une des plus simples et mécaniques pour démontrer formellement la correction d'algorithmes. Pour cela, il faut appliquer les cinq règles suivantes.

1. $\mathbf{WP}(\text{no-op}, Q) \equiv Q$
2. $\mathbf{WP}(x := E, Q) \equiv Q[x := E]$
3. $\mathbf{WP}(C; D, Q) \equiv \mathbf{WP}(C, \mathbf{WP}(D, Q))$
4. $\mathbf{WP}(\text{if } Cond \text{ then } C \text{ else } D) \equiv (Cond = \text{true} \Rightarrow \mathbf{WP}(C, Q)) \wedge (Cond = \text{false} \Rightarrow \mathbf{WP}(D, Q))$
5. $\mathbf{WP}(\text{while } E \text{ do } C \text{ done } \{\text{inv I var V}\}, Q) \equiv I$
Plus les obligations de preuves suivantes :
 - $(E = \text{true} \wedge I \wedge V = z) \Rightarrow \mathbf{WP}(C, I \wedge V < z)$ (preuve que chaque passage décrémente le variant)
 - $I \Rightarrow V \geq 0$ (preuve que le variant reste valide lors les passages successifs)
 - $(E = \text{false} \wedge I) \Rightarrow Q$ (preuve qu'après le dernier passage, Q est bien atteint)

La première règle signifie que la plus faible précondition à assurer pour que Q soit vraie après l'exécution de `no-op` (après l'exécution de rien du tout) est Q elle-même.

La seconde règle explicite les affectations de variable : Pour que Q soit vraie après une affectation, il faut que Q soit vraie au préalable avec une réécriture.

La troisième règle indique que la plus faible précondition permettant que Q soit vraie après l'exécution de `C` puis `D`, c'est la précondition à `C` pour $\mathbf{WP}(D, Q)$ soit vrai, c'est à dire pour que soit vérifiée la précondition de `D` permettant à Q d'être vraie.

La quatrième règle est plus facile à écrire avec l'écriture mathématique qu'avec une paraphrase.

La cinquième règle, au sujet des boucles, n'est pas très compliquée non plus, mais elle impose d'annoter chaque boucle `while` par un invariant et un variant (syntaxe : $\{\text{inv I var V}\}$). Par ailleurs, l'application de cette règle produit trois obligations de preuves supplémentaires. Il s'agit d'expression qu'il faudra démontrer par ailleurs pour avoir le droit d'appliquer la règle de calcul du **WP**.

Ensemble, ces cinq règles permettent de démontrer le triplet de Hoare $\{P\} C \{Q\}$ en montrant la proposition $P \Rightarrow \mathbf{WP}(C, Q)$ ainsi que toutes les obligations de preuves engendrées lors du calcul de $\mathbf{WP}(C, Q)$.

FIB	
i:=1	1
a:=1	2
b:=1	3
while i<n do	4
i:=i+1	5
u:=a	6
a:=a+b	7
b:=u	8
done	9

★ Exercice 1: Fibonacci (d'après Ralf Treinen).

Calculez la plus faible précondition pour que code donné ci-contre admette comme post-condition que $a = fib(n)$ (ie, que l'algorithme calcule Fibonacci de n dans a).

Réponse

On note $Q \equiv a = f_n$.

On applique la règle 3 pour la séquence `l1` d'une part, et tout le reste d'autre part :

$$\mathbf{WP}(\text{FIB}, Q) = \mathbf{WP}(l1, \mathbf{WP}(\text{reste}, Q))$$

Puisque `l1` est une affectation, on applique la règle 2, et on vire gentiment le **WP** englobant.

$$\mathbf{WP}(\text{FIB}, Q) = \mathbf{WP}(\text{reste}, Q) [i := 1]$$

En recommençant 3 fois, on trouve :

$$\mathbf{WP}(\text{FIB}, Q) = \mathbf{WP}(\text{while}, Q) [i := 1; a := 1; b := 1]$$

Il faut maintenant chercher l'invariant et le variant de la boucle. Le variant est $n - i$, de façon à varier entre $z=n$ et $0 (< z)$. L'invariant est $I \equiv 0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1}$

Donc, en appliquant la règle 5, on a : $\mathbf{WP}(\text{FIB}, Q) = I [i := 1; a := 1; b := 1]$, plus les obligations de preuve suivantes :

1. $i < n \wedge I \wedge n - i = z \Rightarrow \mathbf{WP}(loopBody, I \wedge n - i < z)$
2. $I \Rightarrow n - i \geq 0$
3. $(i \geq n \wedge I) \Rightarrow Q$

Pour le premier, on a que des affectations dedans, donc ca se réécrit facilement. Pour les deux autres, remplacer I par sa valeur aide beaucoup :

1. $i < n \wedge I \wedge n - i = z \Rightarrow (I \wedge n - i < z)[i := i + 1, a := a + b, b := a]$
2. $0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1} \Rightarrow n - i \geq 0$
3. $(i \geq n \wedge 0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1}) \Rightarrow a = f_i$

L'obligation de preuve 2 est triviale puisqu'on a bien $0 \leq i \leq n \Rightarrow n - i \geq 0$ (on peut oublier 2 éléments à gauche de \Rightarrow , ce qui était I suffit à prouver ce qui est à droite).

Pareil pour l'obligation de preuve 3 : on ne garde que l'élément au milieu des \wedge , et on a ce qu'il fallait démontrer.

Pour la première, il faut également remplacer I par sa valeur, puis remplacer les résultats d'affectation.

$$i < n \wedge n - i = z \wedge I \Rightarrow (I \wedge n - i < z)[i := i + 1, a := a + b, b := a]$$

On remplace I par sa valeur

$$(i < n) \wedge (n - i = z) \wedge (0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1})$$

$$\Rightarrow (0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1} \wedge n - i < z)[i := i + 1, a := a + b, b := a]$$

On fait les réécritures imposées par les trucs entre crochets. Sans réfléchir comme un cherche/remplace.

$$(i < n) \wedge (n - i = z) \wedge (0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1}) \Rightarrow (0 \leq i + 1 \leq n \wedge a = f_{i-1} + f_{i-2} \wedge b = f_{i-1} \wedge n - (i + 1) < z)$$

a prend cette valeur, car il est l'ancien a plus l'ancien b . L'ancien a est f_{i-1} (vu que i a été mis à jour), et l'ancien b est $f_{(i-1)-1}$ pour la même raison. le nouveau b est l'ancien a

On simplifie l'écriture de part et d'autre du \Rightarrow (drt : $f_i = f_{i-1} + f_{i-2}$; gch, range les éléments sur i)

$$(n - i = z) \wedge (0 \leq i < n) \wedge (a = f_i) \wedge (b = f_{i-1}) \Rightarrow (0 \leq i + 1 \leq n) \wedge (a = f_i) \wedge (b = f_{i-1}) \wedge (n - (i + 1) < z)$$

On a quatre éléments dans la partie droite de cette première obligation de preuve :

- $0 \leq i + 1 \leq n$: on a ça trivialement depuis le $(0 \leq i < n)$ présent à gauche.
- $a = f_i$: présent à l'identique à gauche
- $b = f_{i-1}$: présent à l'identique à gauche
- $n - (i + 1) < z$: À gauche, on a $n - i = z$ donc trivialement $n - i - 1 < z$

Et voila, on a démontré les 3 obligations de preuves. Il ne reste plus qu'à démontrer $P \Rightarrow WP(FIB, Q)$, càd $P \Rightarrow I[i := 1; a := 1; b := 1]$ càd $P \Rightarrow 0 \leq i \leq n \wedge a = f_i \wedge b = f_{i-1}[i := 1; a := 1; b := 1]$ càd $P \Rightarrow 0 \leq 1 \leq n \wedge 1 = f_1 \wedge 1 = f_0$

En virant les éléments trivialement vrais, on trouve qu'il faut démontrer : $P \Rightarrow 1 \leq n$

Mais quelle est donc la proposition P qui fait en sorte que l'expression ci-dessus soit vraie ? Ben $n > 0$ marche très bien, hein. Et voila, par magie de la logique, on vient de montrer que notre fonction calcule bien fibonacci(n) dans sa variable a sous la précondition que $n > 0$

Fin réponse

★ **Exercice 2: Calcul du minimum** (d'après Alexandre Miquel).

On considère le code ci-contre, qui calcule le minimum d'une fonction entre 1 et n .

▷ **Question 1:** Quelle est la spécification formelle de ce programme ? Spécifiez en particulier la post-condition.

▷ **Question 2:** Trouvez l'invariant et le variant de la boucle while.

	MIN	
1	m:=f(1)	
2	i:=2	
3	while i<n do	
4	if (f(i)<m) then m:=f(i)	
5	i++	
6	done	

Réponse

Il faut donc tout d'abord spécifier la post-condition de façon un peu carré de façon à pouvoir le démontrer par la suite. On veut exprimer que m est la plus petite valeur de f sur l'intervalle.

Post $\equiv (\exists u \in [1, n] \text{ tq } m = f(u)) \wedge (\forall v \in [1, n], m \leq f(v))$

On peut également donner la précondition révée (**Pré** $\equiv n > 0$), mais le principe de la preuve avec les WP, c'est justement de calculer la précondition nécessaire pour que ce code donne la post-condition. Donc, en général, la précondition est un résultat de la preuve, pas un truc à deviner. C'est d'ailleurs l'intérêt de la méthode.

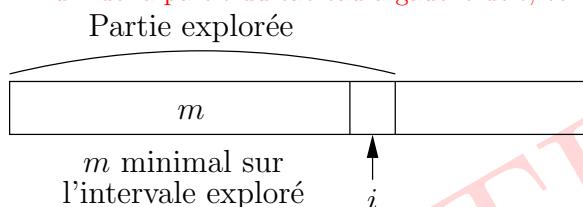
Ensuite, il faut donner le variant. On regarde comment bouge l'indice i , et on tripatouille pour que le variant bouge depuis un nombre non nul au début jusqu'à 0 à la fin. On va donc prendre $V = n - i$

Reste maintenant à deviner l'invariant. Il n'y a pas de méthode mécanique pour cela. Il faut comprendre l'algorithme et voir pourquoi il fonctionne. D'ailleurs, il est important d'insister auprès des élèves sur le fait que l'art de trouver l'invariant est sans doute le bout de ce module qu'ils réutiliseront le plus dans leur vie professionnelle. Quand l'algo est complexe (pas forcément long, mais complexe), il est courant de mettre l'invariant de boucle dans la documentation, pour aider les lecteurs à comprendre ce qui se passe, pourquoi ça fonctionne.

Un indice pour deviner l'invariant ici, c'est qu'on va se retrouver à faire la preuve suivante :

{Pre} Init { $P_{intermédiaire}$ } While {Post}

Pour cela, il serait plus simple si notre invariant ressemblait fortement à la post-condition. Ça tombe bien, car si on regarde l'algo, il est construit pour faire en sorte qu'à n'importe quelle étape i de la boucle, on ait dans m l'élément minimum de la partie du tableau à gauche de i , comme sur le dessin suivant :



L'invariant à prendre pour que ça se passe bien est le suivant. C'est juste la post-condition vérifiée sur les bouts au début du tableau.

Invariant $\equiv (2 \leq i \leq n) \wedge (\exists u \in [1, i] \text{ tq } m = f(u)) \wedge (\forall v \in [1, i], m \leq f(v))$

Fin réponse

▷ **Question 3:** Calculez les obligations de preuve correspondantes, et vérifiez qu'elles sont satisfaites.

Réponse

C'est assez mécanique et un peu répétitif, mais le calcul des **WP** n'est pas très méchant au fond. Ce qui est drôle, c'est qu'ici, on va démontrer que ce code est faux. Encore plus fort : en ne parvenant pas à démontrer la correction du code, on va trouver un cas où il ne fonctionne pas, et on va même en déduire un correctif.

Ne spoilez pas les étudiants, laissez leur la surprise. Le calcul est assez pénible, autant qu'ils aient le choc de son intérêt à plein.

$WP(While, Post) = I \cap (1 \leq i \leq n) \wedge \exists u \in [1; n] : f(u) = m$

$\wedge \forall k \in [1; n], f(k) \geq m$

① $(i < n) \wedge I \cap (n - i = 3) \Rightarrow WP(body, I \cap n - i \leq 3)$

② $I \Rightarrow n - i \geq 0$ trivial avec la première partie de I

③ $(i > n) \wedge I \Rightarrow Post = I_2 \wedge I_3$ quand $i = n$

$\hookrightarrow se et I_1 \Rightarrow i = n$

$WP(body, I \cap n - i \leq 3) = WP(if, I \cap n - i \leq 3)_{[i := i+1]}$

$\equiv f(i) \leq m \Rightarrow WP(m := f(i), I \cap n - i \leq 3)_{[i := i+1]}$ il manque la branche else (qui est un rien)

$\equiv (f(i) \leq m \Rightarrow (I \cap n - i \leq 3)_{[m := f(i)]})_{[i := i+1]}$

$\equiv (f(i+1) \leq m \Rightarrow 2 \leq i+1 \leq n \wedge \exists u \in [1; i+1] : f(u) = m)$

$\wedge \forall k \in [1; i+1], f(k) \geq m$

$\wedge f(i+1) \geq m \Rightarrow 2 \leq i+1 \leq n \wedge \exists u \in [1; i+1] : m = f(u)$

$\wedge \forall k \in [1; i+1], f(k) \geq m$

$\wedge n - i \leq 3$

$i < n \wedge I \cap n - i = 3 \Rightarrow K$ (obligation preuve n°1)

$n - i = 3 \Rightarrow n - i - 1 \leq 2$

$(i < n) \wedge (2 \leq i \leq n) \Rightarrow (2 \leq i+1 \leq n)$

I

$I \text{ et } f(i+1) \geq m \Rightarrow \forall k \text{ du else et } \exists u \text{ du else}$

$I \text{ et } f(i+1) \leq m \Rightarrow \forall k \text{ du then et } \exists u \text{ du then}$

tableau : $\begin{cases} \{n \geq 0\} \\ \{m = f(1)\} \\ \{i=2\} \end{cases} \text{ Init } \{I\}$

réénonciation : $\begin{cases} (2 \leq i \leq n) \wedge (\exists u \in [1; i] : f(u) = m) \\ \wedge (\forall k \in [1; i] : f(k) \geq m) \\ (2 \leq i \leq n) \wedge \exists u \in [1; i] : f(u) = f(i) \\ \wedge \forall k \in [1; i], f(k) \geq m \end{cases}$

En reregardant l'algorithme, on voit tout de suite que la condition du while est fausse : il faut utiliser $i \leq n$ au lieu de $i < n$. Avec cette modification, on peut remodifier dans l'autre sens : on reprend la post-condition qui nous intéresse, ça remet le premier invariant trouvé, et on arrive cette fois à propager le calcul des WP jusqu'au début de l'algorithme, sans anicroche. Dommage, il était 17h59 un vendredi, les étudiants n'ont pas voulu qu'on le fasse proprement pour faire une photo du tableau :)

La preuve se déroule comme prévu, j'ai une post-condition, et je remonte mon code pour calculer la précondition nécessaire. Le code fini par une boucle while, et donc j'applique la règle 5 pour calculer les obligations de preuves avant de pouvoir affirmer que $WP(\text{while}, \text{post}) = \text{inv.}$

Comme d'habitude, les obligations 2 et 3 sont très simples à monter, et la 1 est bien plus dure. D'ailleurs, y'a un ptit bug au tableau puisque pour s'en sortir, il faut considérer que "if (cond) then instr1" est en fait un "if (cond) then instr1 else rien". Ca semble pas important, mais ça change les éléments de preuves. Mais bon, après quelques réécritures, la partie droite de la première obligation de preuve (ie, la WP du corps de boucle) est l'expression K, en bas du tableau.

Le haut de ce tableau-ci porte sur comment montrer que la première obligation de preuve tient, donc. On prend les bouts de K les uns après les autres, et à chaque fois qu'on en a un, on le raye. C'est gros, mais y'a rien de bien sorcier, en fait.

Et sous le trait, maintenant que la dernière obligation est tombée, il ne nous reste plus qu'à finir de calculer les WP en remontant. Mais là, c'est l'échec : On ne peut pas propager le calcul des WP. Avec l'invariant qu'on a choisi (nécessaire pour la post-condition voulue), on se retrouve à dire que pour tout k dans [1,2], $f(k) \geq m$ alors que l'on a testé dans l'initialisation que pour $k=1$, et pas pour $k=2$...

Le tableau porte déjà les séquelles d'une tentative de correction : on a transformé $[1; 2]$ en $[1; 2[$, mais du coup, l'invariant de boucle change en conséquence, ce qui change également la post-condition que l'on trouve. Donc, on a bien montré que si l'élément minimal est dans la dernière position, cet algorithme ne le trouve pas.

Fin réponse**★ Exercice 3: Tri par sélection** (d'après Alexandre Miquel).

On considère le code ci-dessous, qui trie un tableau d'entier.

Dans la logique de Hoare, $t[i]$ ne peut être accédé que si la condition suivante est satisfaite : $0 \leq i < \text{taille}(t)$.▷ **Question 1:** Réécrivez ce programme en changeant les boucles **for** en boucles **while** équivalentes.▷ **Question 2:** Montrez la correction du code de la permutation circulaire :

$$\{t[i] = x \wedge t[j] = y\} \quad \text{tmp} := t[i]; t[i] := t[j]; t[j] := \text{tmp} \quad \{t[i] = y \wedge t[j] = x\}$$

▷ **Question 3:** Montrez à l'aide de ce qui précède la correction de l'algorithme du tri par sélection.**Réponse**

Pour de vrai, je n'ai jamais calculé le WP de cet algorithme-ci.

Fin réponse

SEL

```
1 for i := 0 to n-2 do
2   for j := i+1 to n-1 do
3     if t[j] < t[i] then
4       tmp := t[i]
5       t[i] := t[j]
6       t[j] := tmp
7     end
8   done
9 done
```