INGI1122 – Méthodes de conception de programmes
Séance 1 – Axiomes et Logique de Hoare

Guillaume Maudoux, Xavier Gillard et Charles Pecheur

Dans les exercices suivants, toutes les variables sont de type entier. On raisonne sur les entiers illimités, sans tenir compte des problèmes de dépassement.

Exercice 1. *Triplets de Hoare.* Pour rappel, un triplet de Hoare $[P]$ `S` $[Q]$ est une proposition logique valide ou invalide. Un triplet est valide si et seulement si, si P est vrai avant l'exécution de `S`, alors l'exécution se termine et Q est vrai après.

Déterminez la validité de chacun des triplets suivants ($\mathsf{abs}(x)$ calcule la valeur absolue de x).

Rendez-vous sur `www.wooclap.com/TPMCP` !

a. $[i = 1]$ `j := i;` $[i = j = 1]$ — Valide

b. $[x > 0]$ `x,y := y,x;` $[y \geq 0]$ — Valide

c. $[a > 0 \wedge b < 0]$ `c := a * b;` $[c \geq 0]$ — Invalide

d. $[0 \leq t < N]$ `t := t+1;` $[0 < t \leq N]$ — Valide

e. $[m = 2 \wedge n = 10 \wedge o = m^n]$ `n := m;` $[o = 4]$ — Invalide

f. $[p = -1]$ `p := abs(p);` [false] — Invalide

g. $[p = -1]$ `p := abs(p);` [true] — Valide

h. [true] `p := abs(p);` $[p > 0]$ — Invalide

Soit la fonction $\mathsf{fib}(n)$ qui calcule le n-ième nombre de Fibonacci pour $n \geq 0$ et ne se termine pas si $n < 0$. Déterminez la validité de chacun des triplets suivants.

i. [true] `p := fib(p);` [true] — Invalide

j. [false] `p := fib(p);` $[p < 0]$ — Valide

k. $[a * b > 0]$ `p := fib(a+b);` $[p \geq 0]$ — Invalide

l. $[a \geq b]$ `p := fib(a-b);` $[p \geq 0]$ — Valide

Exercice 2. *Variables auxiliaires.* Les variables auxiliaires, aussi dites fantômes ou rigides, sont des variables qui n'apparaissent pas dans le programme en cours de vérification et dont la valeur ne change donc pas durant l'exécution de ce dernier.

a. Complétez le triplet suivant afin d'exprimer que la valeur de n a été doublée.

[?] `n := n * 2;` [?] [X=n] n := n * 2; [n=X*2]

b. Complétez le triplet suivant afin qu'il soit valide, qu'il exprime que la valeur de $a+b$ reste inchangée et que b ne devient jamais négatif.

[?] `a, b := a+1, b-1;` $[b \geq 0 \wedge ?]$ [a+b = M ∧ b > 0] a,b := a+1, b-1; [b >= 0 ∧ a+b = M]

c. Prouvez que ces deux triplets sont valides.

Exercice 3. *Affectations.* Pour valider la spécification d'une affectation, on procède en marche arrière par application de l'*axiome de l'affectation.*

Annotez le code suivant à l'aide d'assertions afin d'obtenir un tableau complet (contenant au moins une assertion entre chaque instruction) et qui démontre que le code permute la valeur des trois variables (`a, b, c := b, c, a`). Prouvez ensuite chaque étape.

[a =A & b = B & c = C]

```
a := a + b + c;      b = B & c = C & a = A
c := a - b - c;      b = B & c = C & (a-b-c) = A
b := a - b - c;      b = B & (a-b-c) = C & c = A
a := a - b - c;      [ (a-b-c) = B & b=C & c=A ]
```

[a = B & b = C & c = A]

Exercice 4. *Conditions.* La règle de preuve pour les conditions consiste à prouver la branche vraie en s'aidant du fait que le test est vrai, et la branche fausse en s'aidant du fait que le test est faux. Dans les deux branches, la postcondition à établir est celle de la condition elle-même.

a. On cherche à ordonner deux variables x et y de sorte que x ne soit pas plus grand que y. Pour ce faire, nous allons utiliser une condition `if x > y then ... else ... end`. Spécifiez l'algorithme complet et calculez les affectations nécessaires dans chaque branche de la condition.

b. On cherche à calculer $y = \mathsf{abs}(x - 5)$ pour une entrée x donnée avec le programme suivant : `if x-5 > 0 { y := x-5; } else { y := 5-x; }`. Donnez une spécification complète du problème, et prouvez que le programme est correct.

Exercice 5. *Itérations.* La multiplication matricielle implique un calcul de sommes de produits d'éléments.

```
i := 0;
while i < n {
  j := 0;
  while j < n {
    k := 0;
    while k < n {
      C[i,j] := C[i,j] + A[i,k] * B[k,j];
      k := k + 1;
    }
    j := j + 1;
  }
  i := i + 1;
}
```

En partant de la boucle la plus profondément imbriquée, donnez pour chaque boucle un invariant et un variant valides. Ensuite, prouvez la validité de vos assertions pour la boucle qui itère sur k.

Exercice 6. *Exponentielle* (bonus). Dans cet exercice, on vous demande de réaliser la preuve d'une partie d'un programme qui calcule l'exponentielle $r = a^b$ de deux nombres a et b.

a. Prouvez le triplet suivant par application de l'axiome d'affectation. Synthétisez vos résultats dans un tableau reprenant vos propositions intermédiaires.
$[r * a^b = E]$ `r, b := r * a, b - 1;` $[r * a^b = E]$

b. Prouvez le triplet suivant par l'axiome d'affectation ($\mathsf{pair}(b)$ signifie que b est pair). Attention, en arithmétique entière, il faut que b soit pair pour que $(a^2)^{b/2} = a$.
$[r * a^b = E \wedge \mathsf{pair}(b)]$ `a, b := a * a, b / 2;` $[r * a^b = E]$

c. Vous venez de prouver les deux parties de l'algorithme d'exponentiation rapide. On vous demande maintenant d'annoter le code suivant pour exprimer qu'il conserve l'invariant $r * a^b = E$.
Assurez-vous que vos assertions sont suffisantes pour vérifier le programme.

```
if b \% 2 == 0 {
  a, b := a * a, b / 2;
} else {
  r, b := r * a, b - 1;
}
```

d. Pour finir l'algorithme, ajoutez une boucle `while` et les invariants nécessaires pour prouver la validité du programme.

Devoir *A soumettre sur Moodle pour la prochaine séance.*

On a deux tableaux a et b de même taille $|a| = |b| = N$ (indicés de 0 à $N - 1$). Soit le programme suivant qui remplit b de sorte que $b[i] = a[0] + a[1] + \cdots + a[i]$. Écrivez la spécification complète du programme (préconditions, postconditions et invariants).

```
b[0] := a[0];
```

```
var i := 1;
while i < N {
    b[i] := a[i] + b[i-1];
    i := i + 1;
}
```
