TD3 : Preuves de terminaison et de correction

Exercice 1 (Division enclidienne) On considère l'algorithme suivante :

Algorithme 1 Division euclidienne

entrée : entiers naturels a et b, avec $b > 0$
sortie : quotient q, reste r

1 : $q \leftarrow 0$; $w \leftarrow b$; $r \leftarrow a$
2 : **tant que** $w \leq r$ **faire**
3 : $\quad w \leftarrow 2 * w$
4 : **fin tant que**
5 : **tant que** $w \neq b$ **faire**
6 : $\quad q \leftarrow 2 * q$
7 : $\quad w \leftarrow w/2$ /* division entre entiers */
8 : $\quad$ **si** $w \leq r$ **alors**
9 : $\quad\quad r \leftarrow r - w$
10 : $\quad\quad q \leftarrow q + 1$
11 : $\quad$ **fin si**
12 : **fin tant que**
13 : renvoie (q, r)

(je ne mets pas la fonction `C` parce que vous ne savez pas encore comment renvoyer un couple de valeurs en `C`)

1. Faites tourner la fonction à la main sur les valeurs `(10,4)`, puis sur les valeurs `(15,5)` pour trouver ce que fait cette fonction.
2. Montrer que l'appel à la fonction se termine quelles que soient les entrées (qui vérifient les pré-conditions indiquées), en utilisant des variants de boucle.
3. Montrer que la fonction calcule effectivement ce que votre intuition vous a indiqué lors de la question 1 (si vous êtes en panne d'intuition : la fonction est documentée), en utilisant un invariant de boucle.

Exercice 2 (Tri bulle) On donne la fonction suivante qui exécute le *tri bulle*.

```c
/**
 * entree : un tableau d'entiers et sa taille
 * sortie : le tableau est trie en place
 */
void tri_bulle(int *tab, int n){
  bool permutation = true;
  while(permutation){ // si le dernier passage a fait au moins une modification
    permutation = false;
    for(int i=0; i<n-1; i++){
      if(tab[i] > tab[i+1]){ // si ce n'est pas dans le bon ordre, on inverse
        int tmp = tab[i];
        tab[i] = tab[i+1];
        tab[i+1] = tmp;
        permutation = true;
      }
    }
    n = n - 1;
  }
}
```

1. Montrer que l'exécution de la fonction `tri_bulle` se termine quelle que soit l'entrée (en supposant que `n` est bien la longueur du tableau `tab`).
2. Montrer qu'à la fin de l'exécution de la fonction `tri_bulle`, les données du tableau sont triées. Pour cela, vous pourrez trouver un invariant qui explicite le nombre de données à leur place finale après chaque itération de la boucle `while`.

Exercice 3 (Produit) On donne la fonction suivante :

```
1  /** a, b : entiers positifs
2      resultat : a*b */
3  int multiplication(int a, int b){
4    int m = 0;
5
6    assert(a>=0);
7    assert(b>=0);
8
9    while(a>0){
10     if(a%2 == 1){
11       m = m+b;
12     }
13     a = a/2;
14     b = b*2;
15   }
16
17   return m;
18 }
```

1. Faites tourner la fonction `multiplication` à la main sur les valeurs 13 et 15 et constater qu'elle semble faire ce qu'elle prétend faire.
2. Montrer que l'appel à la fonction `multiplication` se termine quelles que soient les entrées, en utilisant un variant de boucle.

   Correction :
   La valeur initiale de la variable entière `a` est strictement positive par hypothèse et sa seule affectation est celle de la ligne 13, donc la valeur de a est toujours positive. Si cette valeur est nulle, on sort de la boucle, donc l'affectation de la ligne 13 se fait toujours avec une valeur strictement positive avant la division et la valeur de `a` diminue strictement à chaque itération.
   L'expression `a` est donc un variant pour la boucle `while` de la ligne 9, ce qui nous assure qu'on sort de la boucle.

3. Montrer que le fonction calcule effectivement le produit des deux arguments qui lui sont transmis, en utilsant un invariant de boucle.

   Correction :
   Notons a_0, b_0, m_0 les valeurs de a, b, m avant la boucle et a_i, b_i, m_i leurs valeurs après la ie itération (la première itération étant numérotée 1). En particulier $m_0 = 0$.
   Montrons qu'on a l'invariant **I** suivant : $a_i b_i + m_i = a_0 b_0$.
   Avant la boucle : on a $a_i b_i + m_i = a_0 b_0 + m_0 = a_0 b_0$, donc **I** est vérifié.
   Supposons **I** vérifié à l'entrée de la ie itération, soit $a_{i-1} b_{i-1} + m_{i-1} = a_0 b_0$.
   Alors à la fin de la ie itération, on a :
   - Si a_{i-1} est pair, notons $a_{i-1} = 2k$ pour un certain k. Alors $a_i = k$, $b_i = 2b_{i-1}$ et $m_i = m_{i-1}$, donc :
     $$a_i b_i + m_i = k \times 2b_{i-1} + m_{i-1} = a_{i-1} b_{i-1} + m_{i-1} = a_0 b_0$$
     par hypothèse ;
   - Si a_{i-1} est impair, notons $a_{i-1} = 2k+1$ pour un certain k. Alors $a_i = k$, $b_i = 2b_{i-1}$ et $m_i = m_{i-1} + b_{i-1}$, donc :
     $$a_i b_i + m_i = k \times 2b_{i-1} + m_{i-1} + b_{i-1} = a_{i-1} b_{i-1} + m_{i-1} = a_0 b_0$$
     par hypothèse.

   L'invariant **I** est donc bien vérifié en sortie d'itération s'il l'est en entrée.
   Comme on sait qu'on sort de la boucle (question précédente), on peut en déduire que l'invariant reste vérifié en sortie de boucle, soit
   $$a_\infty b_\infty + m_\infty = a_0 b_0$$
   si on note ∞ les indices de sortie de boucle. Or en sortie de boucle on a $a_\infty = 0$, on en déduit donc $m_\infty = a_0 b_0$. La fonction `produit` calcule bien le produit de ses deux arguments.

Exercice 4 (Exponentiation rapide) On considère la fonction d'exponentiation rapide :

```
1  /** a : entier
2       n : entier positif ou nul
3       sortie : a puissance n
4  */
5  int puissance(int a, int n){
6    int p = 1;
7    while (n > 0) {
8      if (n%2 == 1){
9        p = p * a;
10     }
11     a = a * a;
12     n = n / 2;
13   }
14
15   return p;
16 }
```

1. Dérouler cette fonction sur l'entrée $(2,5)$.
2. Donner un variant de boucle qui permet de conclure que cette fonction se termine.

Correction :
L'expression n est un vairant de boucle pour la boucle `while` de la ligne 7 : si $n=0$, on ne rentre pas dans la boucle, sinon n est un entier strictement positif, et sa valeur est divisée par deux à chaque itération (ligne 12), elle décroît donc strictement. Toutes les instructions s'exécutant en temps fini, on en déduit que l'exécution de la boucle se termine, ainsi que l'exécution de la fonction.

3. Montrer grâce à un invariant de boucle que cette fonction fait bien ce qu'on attend d'elle.

Correction :
Soit le prédicat

$$(I) \qquad p \times a^n = a_0^{n_0},$$

où a_0 et n_0 sont les valeurs respectives de a et n avant le début de la boucle.
Montrons que (I) est un invariant de boucle pour la boucle `while` de la ligne 7.
Avant l'exécution de la boucle, on a $p=1$, $a=a_0$ et $n=n_0$, donc $p \times a^n = a_0^{n_0}$, donc (I) est vérifié.
Supposons que (I) soit vérifié au début d'une certaine itération. Notons a, n, p les valeurs respectives de `a`, `n` et `p` au début de l'itération et a', n', p' leurs valeurs à la fin de l'itération. On a donc

$$p \times a^n = a_0^{n_0},$$

et on obtient

si n est pair :

$$p' = p, \quad a' = a^2 \quad \text{et} \quad n' = \frac{n}{2}$$

on en déduit

$$p' \times a'^{n'} = p \times (a^2)^{\frac{n}{2}} = p \times a^n = a_0^{n_0}.$$

si n est impair :

$$p' = pa, \quad a' = a^2 \quad \text{et} \quad n' = \frac{n-1}{2}$$

on en déduit

$$p' \times a'^{n'} = pa \times (a^2)^{\frac{n-1}{2}} = p \times a^n = a_0^{n_0}.$$

Le prédicat (I) est donc vérifié en fin d'itération. On en déduit que c'est un invariant pour la boucle.
Comme on sait qu'on sort de la boucle (question précédente), le prédicat (I) est vérifié en sortie de boucle, on obtient alors :

$$p \times a^n = a_0^{n_0}.$$

Or en sortie de boucle, on a $n=0$, ce qui permet d'obtenir la valeur renvoyée par la fonction :

$$p = a_0^{n_0}.$$

La fonction possède donc une correction totale.