Chapitre :
Preuve d'algorithmes

64

Preuve d'un algorithme

- La correction (ou preuve) d'un algorithme consiste à démontrer que celui-ci fonctionne, c'est-à-dire :
  - Répond-il correctement à la question ?
  - Ne boucle-t-il pas indéfiniment ?
- La correction d'un algorithme correspond donc à deux critères :
  - Correction partielle : l'algorithme calcule le bon résultat
  - Terminaison : l'algorithme répond en temps fini
- Faire la preuve d'un algorithme consiste ainsi à prouver que pour toute entrée vérifiant sa précondition :
  - Il produit une sortie vérifiant sa postcondition (correction partielle) ;
  - Et ce en temps fini (terminaison).

65

Les propriétés recherchées

Terminaison

L'exécution de l'algorithme produit-elle un résultat en temps fini **quelles que soient** les données fournies ?

Correction partielle

Lorsque l'algorithme s'arrête, le résultat calculé est-il la solution cherchée quelles que soient les données fournies ?

Terminaison + correction partielle = correction totale

Quelles que soient les données fournies, l'algorithme s'arrête et donne une réponse correcte.

Pour certains problèmes il n'existe **que** des algorithmes **partiellement** corrects !

66

Exercice

Quel(s) programme(s) calcule(nt) la factorielle de n dans la variable F ?

A
```
i := 0
F := 1
while i <= n
    i := i + 1
    F := F * i
```

C
```
i := 1
F := 1
while i <= n
    F := F * i
    i := i + 1
```

B
```
i := 0
F := 1
while i < n
    i := i + 1
    F := F * i
```

D
```
i := 0
F := 1
while i < n
    F := F * i
    i := i + 1
```


67

Correction d'un algorithme

- Pour prouver qu'un algorithme termine, il suffit de montrer qu'il ne boucle pas à l'infini.
  - Toute algorithme sans appel de fonction ni répétitive termine ;
  - Toute répétitive pour *itère* un nombre fini de fois ;
  - Une répétitive `tant que` (et les appels récursifs) peut boucler à l'infini si sa condition reste toujours vraie.
- Ainsi, pour prouver la terminaison d'un algorithme il faudra montrer que les répétitives `tant que` ne bouclent pas à l'infini, c'est-à-dire que leurs conditions deviennent fausses au bout d'un certain nombre d'itération.

Définitions

Soit un problème instancié par une donnée D et dont la réponse est fournie par un résultat R.

Une spécification peut être donnée sous la forme de :

- une propriété $P(D)$ de la donnée *(précondition)* ;
- une propriété $Q(D, R)$ de la donnée et du résultat *(postcondition)*.

Un programme satisfait cette spécification si :

Pour toute donnée D qui vérifie la propriété P,
l'exécution du programme donne un résultat R qui vérifie $Q(D, R)$.

Le programme est alors dit correct *par rapport à* cette spécification.

Notion d'invariant

Idée de la démonstration d'un algorithme

De proche en proche, établir que la postcondition est vraie à chaque fois que la précondition est vraie.

- Affectation, séquence, condition :
  pas de vrai problème si la spécification est correctement écrite.
- Problème : la boucle
  (peut recevoir ses données d'une itération précédente)

Invariant

Un invariant est une propriété P des variables en début de boucle telle que

si P est vérifiée à une itération, alors elle l'est à l'itération suivante.

Méthodologie

1. **Choisir et exprimer** un invariant *judicieux*
   Pas de méthode systématique
2. **Démontrer** qu'il est vérifié avant d'entrer dans la boucle
   Utiliser les préconditions
3. **Démontrer** que s'il est vérifié au début d'une itération quelconque, il l'est aussi au début de l'itération suivante.
   Utiliser le corps de la boucle
   On note x' la valeur de x en fin de boucle
4. **Instancier** l'invariant en sortie de boucle et en déduire une postcondition.
   *Utiliser (la négation de) la condition du **while***

Exemple : division euclidienne

DIV(a, b)
Données : Deux entiers a et b
Résultat : Le quotient q et le reste r de la division euclidienne de a par

Précondition : $a \geq 0$ et $b > 0$
$r \leftarrow a$
$q \leftarrow 0$
tant que $r \geq b$

$r \leftarrow r - b$ {invariant : a = b × q + r}
$q \leftarrow q + 1$
{b×q' +r' = b×(q+1)+(r−b) = b×q+b−b+r = b×q+r = a}

retour q, r
Postconditions : a=b×q+r
0≤r<b
a et b inchangés

72

Exercice

Quel(s) programme(s) calcule(nt) la factorielle de n dans la variable F ?

A
```
i := 0
F := 1
while i <= n
    i := i + 1
    F := F * i
```

C
```
i := 1
F := 1
while i <= n
    F := F * i
    i := i + 1
```

B
```
i := 0
F := 1
while i < n
    i := i + 1
    F := F * i
```

D
```
i := 0
F := 1
while i < n
    F := F * i
    i := i + 1
```


73

Programme A

i := 0
F := 1 $\{F = 1 = 0! = i!\}$
while $i <= n$
$\{F = i!\}$
i := i + 1
F := F * i
$\{F' = F * i' = F * (i+1) = i! * (i+1) = (i+1)!$ donc $F' = i'!\}$

Invariant

$F = i!$

Mais en sortie de boucle $i = n+1$ d'où $F = (n+1)!$

74

Programme B

i := 0
F := 1 $\{F = 1 = 0! = i!\}$
while $i < n$
$\{F = i!\}$
i := i + 1
F := F * i $\{F' = F * i' = i! * (i+1) = (i+1)!$ donc $F' = i'!\}$

Invariant

$F = i!$

En sortie de boucle $\{i = n$ d'où $F = n!\}$

75

Programme C

```
i := 1
F := 1    {F = 1 = 0! = (1 − 1)! = (i − 1)!}
while i <= n
    {F = (i − 1)!}
    F := F * i
    i := i + 1    {F' = F ∗ i = (i − 1)! ∗ i = i! donc
                   F' = (i + 1 − 1)! = (i' − 1)!}
```

Invariant

$F = i!$ ~~$F = i!$~~ alors en fin d'itération $F' = F * i = i! * i \neq i'!$: NON

$$F = (i-1)!$$

En sortie de boucle i = n+1 d'où F= n !

76

Programme D

```
i := 0
F := 1
while i < n
    F := F * i
    i := i + 1
```

Invariant

$F = (i-1)!$ ~~$F = (i-1)!$~~ correctement propagé par la boucle
mais faux à l'entrée de la boucle : NON

En réalité on a toujours $F = 0$ après la 1è itération.

77

Correction partielle ou totale

Correction partielle

Pour toute donnée D qui vérifie la précondition P,
si le programme se termine,
alors son exécution donne un résultat R qui vérifie $Q(D, R)$.

Correction totale

Pour toute donnée D qui vérifie la précondition P,
l'exécution du programme se termine
et donne un résultat R qui vérifie $Q(D, R)$.

$$\text{Correction partielle} \wedge \text{terminaison} \Rightarrow \text{Correction totale}$$

D'où l'idée de prouver la terminaison **en même temps** que la correction partielle : on enrichit les annotations existantes.

78

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \ldots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;
- v (en entrée de boucle) soit toujours positive ;
- v décroît strictement.

Théorème 2.2 (Terminaison d'une boucle, d'un algorithme)

1. Une boucle possédant un variant de boucle se finit
2. Un algorithme dont toutes les boucles possèdent un variant se finit.

Avertissement 2.3

Doit être valable pour toute entrée possible.

79

Variant de la division euclidienne

Division par soustractions

DIV(a, b)
Données : Deux entiers a et b
Résultat : Le quotient q et le reste r de la division euclidienne de a par b
Précondition : $a \geq 0$ et $b > 0$
$r := a$
$q := 0$
while $r \geq b$ $\quad \{\mathtt{r} = n\}$
  $r := r - b$
  $q := q + 1$ $\quad \{0 \leq \mathtt{r} < n\}$
return q, r

- r est clairement un variant de boucle
- On peut le formuler dans les annotations existantes grâce à une nouvelle variable logique.
  (n est quantifiée existentiellement de façon implicite)
- La preuve de $\mathtt{r} < n$ en fin de boucle repose sur la précondition $b > 0$ et sur la condition du `while`.

80

Les difficultés

- Précondition et postcondition
  - généralement faciles à écrire si le problème est correctement spécifié
  - éventuellement nécessaire de renforcer la précondition si la preuve n'aboutit pas
  - attention aux postconditions trop faibles
- Invariants
  - incluent souvent une généralisation de la postcondition I demandent une compréhension fine de l'algorithme
  - les trouver peut même précéder l'écriture de l'algorithme
- Variants
  - souvent immédiats, mais des cas particuliers très difficiles I nécessitent parfois de s'appuyer sur les autres assertions

81

Exemple

- Spécification :
  - Entrées : tableau T de n entiers
  - Sortie : entier maxi
  - Rôle : chercher le maximum de T
  - Précondition : T est non vide
  - Postcondition : maxi est l'élément maximal de T

82

Exemple

```
def maximum(T):
    maxi = T[0]
    for i in range(1, len(T)):
        if T[i] > maxi:
            maxi = T[i]
    return maxi
```

- Prouver la terminaison de l'algorithme de recherche du maximum précédent est facile.
- En effet, cet algorithme ne contient qu'une boucle `pour` qui termine nécessairement.

83

Exemple 2

- Considérons l'algorithme de recherche séquentielle d'une occurrence
- ```
  def appartient(v, T):
      i = 0
      trouvee = False
      while i < len(T) and trouvee == False:
          if T[i] == v:
              trouvee = True
          i = i + 1
      return trouvee
  ```
- Il s'agit de trouver un variant de la boucle while, c'est-à-dire une quantité entière qui diminue strictement à chaque itération tout en restant positive.

Remarques

Remarque 2.5

- La terminaison assure un arrêt théorique de l'algorithme.
- Cela ne tient pas compte des problèmes de complexité : un arrêt théorique en quelques milliards d'années est d'un intérêt limité.

Remarque 2.6

Dans le cas d'une boucle `Pour` $i \leftarrow a$ `à` b, un variant simple est $b - i$. Tout boucle `for` se finit.
  ```

Exercice

Algorithme 19 : Mystère

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant b$ **faire**
    $r \leftarrow r - b$;
    $q \leftarrow q + 1$;
  fin tant que
  renvoyer q,r
fin si

86

Exercice

```
def puissance(k, n):
"""Calcule k^n pour deux entiers k, n >= 0."""
i = n
res = 1
while i > 0:
    res = k * res
    i -= 1
return res
```


87

Exercice

```
def rang_dernier_strictement_positif(u_0):
u = u_0
n = 0
while u > 0:
    u = u/2 - 3*n
    n += 1
return n - 1
```


88

Terminaison d'un algorithme

- Deux questions peuvent se poser :
  - L'algorithme s'arrête-t-il ? (problème de terminaison)
  - L'algorithme renvoie-t-il le résultat attendu ? (problème de correction)

89

Correction d'un algorithme

- Montrer que la sortie produite vérifie la postcondition de l'algorithme (quelle que soit l'entrée vérifiant la précondition).
- Utilisation de la notion d'invariant de boucle.
  - propriété attachée à une boucle qui :
    - Est vraie initialement, avant de commencer la boucle ;
    - Est maintenue vraie par toute itération de la boucle, d'où son nom d'invariant.

Correction d'un algorithme

- Méthode : 3 étapes pour prouver la correction partielle d'un algorithme :
  - Etape 1 : INITIALISATION
    - On prouve que l'invariant est vrai avant l'entrée dans la boucle et donc avant d'exécuter la première itération.

Correction d'un algorithme

- Méthode 3 étapes pour prouver la correction partielle d'un algorithme :
  - Etape 2 : CONSERVATION
  - On prouve que l'invariant est conservé par une itération de boucle.
  - Pour cela : on suppose que l'invariant est vrai avant l'itération i de boucle puis on montre que l'invariant est toujours vrai après l'itération i (et donc vrai avant l'itération i+1).

92

Correction d'un algorithme

- Méthode 3 étapes pour prouver la correction partielle d'un algorithme :
  - Etape 3 : CONCLUSION
    - On utilise le fait que l'invariant soit vrai en sortie de boucle pour montrer la correction partielle de l'algorithme.

93

Correction d'un algorithme

- Exemple

```
def maximum(T):
    maxi = T[0]
    for i in range(1, len(T)):
        if T[i] > maxi:
            maxi = T[i]
    return maxi
```


94

Correction d'un algorithme

- Exercice

```
def multiplication(a,b):
p = 0
i = 0
while i < a:
    p = p + b
    i = i + 1
return p
```


95

Correction d'un algorithme

- Exercice

```
def truc(n):
    p=2
    q=1
    for i in range(n):
        q=q*p
        p=p+1
    return q
```


96

Correction d'un algorithme

- Exercice

```
a = 100
while a >= 0:
    a = a//2
```


97