Correction des programmes

Nous nous sommes demandés jusqu'à présent si un programme termine. Pour le moment, ce que nous ne savons pas faire, c'est montrer qu'un programme calcule effectivement ce que l'on lui demande. Un programme prend en entrée un certain nombre de données et retourne de nouvelles données. L'étude de la *correction partielle* d'un programme – on dit aussi *la preuve du programme* – c'est l'étude de l'expression des sorties en fonction des entrées afin de vérifier que si les entrées sont conformes, les sorties ont bien les valeurs attendues. Pour réaliser une étude de correction, on exprime à chaque étape du programme les valeurs des variables, c'est-à-dire qu'on décrit symboliquement *l'état mémoire*. L'étude de la *correction partielle* d'un programme, c'est faire la preuve que le programme effectue bien ce qu'on attend de lui. Si on a également montré la terminaison, on dit que le programme est *totalement correct*.

Définition 1 : correction partielle et correction totale

correction totale = correction partielle + terminaison

I Premiers exemples simples

La preuve de programme est simple tant qu'on n'a pas affaire à des boucles : il suffit d'exécuter le programme avec des valeurs *symboliques* pour les entrées. Comme le nombre de chemins d'exécution est fini et que chaque exécution est finie, on peut calculer finiment ce qu'effectue le programme.

1 – Montrer que l'algorithme suivant échange les valeurs des variables x et y. Il faut montrer que si $x = X$ et $y = Y$ avant exécution de l'algorithme alors après exécution, $x = Y$ et $y = X$.

Algorithme 1 :

$t \leftarrow x$;
$x \leftarrow y$;
$y \leftarrow t$;

2 – Montrer que l'algorithme suivant réalise la même tâche. Quel est son avantage ? Quels sont ses inconvénients ?

Algorithme 2 :

$x, y : entier$;
$x \leftarrow x + y$;
$y \leftarrow x - y$;
$x \leftarrow x - y$;

3 – Montrer que si la contrainte sur l'entrée est respectée, alors l'algorithme suivant renvoie 0.

Algorithme 3 :

Entrées : un entier x tel que $1 \leq x \leq 3$
Sorties : l'entier 0
y : *entier*;
si $x == 1$ **alors**
| $y \leftarrow x - 1$
sinon si $x == 2$ **alors**
| $y \leftarrow x - 2$
sinon
| $y \leftarrow x - 3$
fin
retourner y

4 – Montrer que l'algorithme suivant renvoie le maximum entre les deux entrées.

Algorithme 4 :

Entrées : deux entiers x et y
Sorties : le plus grand entier des deux
max : *entier*;
si $x \leq y$ **alors**
| $max \leftarrow y$
sinon
| $max \leftarrow x$
fin
retourner max

5 – Montrer que l'algorithme suivant calcule la valeur absolue de l'entrée.

Algorithme 5 :

Entrées : un entier x
Sorties : la valeur absolue de x
abs : *entier*;
si $x < 0$ **alors**
| $absx \leftarrow -x$
sinon
| $absx \leftarrow x$
fin
retourner $absx$

II Cas des boucles `Tant Que`

La présence d'une boucle peut engendrer une exécution symbolique infinie. Il faut donc mettre en place une démarche spécifique pour traiter ce cas, que nous détaillons maintenant. Nous nous limitons ici à des boucles `Tant Que`, ce qui inclut le cas des boucles `Pour`.

6 – Montrer qu'on peut toujours ré-écrire une boucle `Pour` en boucle `Tant Que` équivalente. Appliquer cette transformation à l'algorithme suivant.

Algorithme 6 :

Entrées : un entier n
Sorties : un entier
$somme$: *entier* $\leftarrow 0$;
pour i : *entier de* 1 *à* n
faire
| $somme \leftarrow somme + 1$;
fin
retourner $somme$

Le schéma de programme que nous examinons est le suivant :

Algorithme 7 : schéma de programme

Entrées : pré-condition P_{entree}
Sorties : post-condition P_{sortie}
Pré-traitement
tant que B **faire**
| Corps de boucle
fin
Post-traitement

Montrer la correction partielle de ce schéma de programme, c'est montrer que :
— si les entrées vérifient la pré-condition P_{entree},
— alors les sorties vérifient la post-condition P_{sortie}.
Le couple (pré-condition, post-condition) est un contrat sur l'emploi de l'algorithme. Si les entrées ne vérifient pas la pré-condition, l'algorithme n'est pas dans son champ d'application et tout peut arriver.

On résume la boucle `Tant Que` par un *invariant de boucle* I. Un invariant de boucle est une formule logique portant sur les variables du programme, par exemple $x > 0 \vee (y \leq 1 \wedge z > 2)$). On montre que I est vrai au point de programme (*) du programme ci-dessous. C'est-à-dire qu'à chaque fois que l'exécution du programme passe au point (*), l'état mémoire courant – les valeurs des variables à ce moment là – valide I. La méthode pour justifier la validité de I est décrite dans la définition et le schéma de programme ci-dessous. Juste après la boucle, on sait que I est vraie, ainsi que la négation du test de la boucle, car justement on est sorti de la boucle. On a donc en ce point $I \wedge \neg B$. Enfin, après un éventuel post-traitement, on montre que l'état mémoire courant implique P_{sortie}.

Algorithme 8 : schéma de programme avec les obligations de preuve dûment signalées

Entrées : pré-condition P_{entree}
Sorties : post-condition P_{sortie}
Pré-traitement
% *cas de base :*
% *on montre que* I *est vrai avec la pré-condition* P_{entree} *et le pré-traitement.* À justifier.
(*) **tant que** B **faire**
| % *cas inductif :*
| % *on suppose que* I *et* B *sont vrais.*
| Corps de boucle
| % I *reste-t-il vrai ?* À justifier.
fin
% *ici, on sait que* I *est vrai et* B *est faux.*
Post-traitement
% *les sorties valident-elles la post-condition* P_{sortie} *?* À justifier.

Pour montrer un invariant de boucle, l'idée est de mettre en place une démonstration par récurrence "sous contrainte". Si on prime les variables pour dénoter leurs valeurs à la fin du corps de la boucle, par exemple x' est la valeur de x à la fin du corps de la boucle, le cas inductif ne consiste pas à montrer que $\forall[I(x, y \ldots) \implies I(x', y', \ldots)]$ mais que $\forall[I(x, y \ldots) \wedge B \implies I(x', y', \ldots)]$. En effet, il faut tenir compte du fait que si l'exécution démarre au début du corps de la boucle, non seulement l'invariant I est vrai mais le test B du `Tant Que` aussi.

Définition 2 : invariant de boucle

On appelle la propriété I décrite précédemment un *invariant de boucle* au point de programme (*). Pour montrer que l'invariant I est vrai en (*), on justifie que :
— I est vrai la première fois que l'exécution passe au point (*) ;
— si $I \wedge B$ est vrai au tout début de la boucle, alors I reste vrai à la fin de la boucle.

7 – Montrer que dans l'algorithme suivant, la propriété I : $somme = \sum_{k=0}^{i} k \wedge i < n+1$ est un invariant de boucle. En déduire que *si il termine*, l'algorithme retourne bien la somme des n premiers entiers. Montrer que l'algorithme termine en justifiant une fonction de rang.

Algorithme 9 :

Entrées : un entier naturel n
Sorties : un entier, la somme $1 + \ldots + n$
i : *entier* $\leftarrow 0$;
$somme$: *entier* $\leftarrow 0$;
(*) **tant que** $i < n$ **faire**
  $i \leftarrow i + 1$;
  $somme \leftarrow somme + i$;
fin
retourner *somme*

L'invariant de boucle doit être utile à la preuve de programme ! Il n'y a pas unicité de l'invariant et en général, il n'existe pas d'algorithme pour déterminer l'invariant *adéquat* ou le *meilleur* invariant. Le choix du *bon* invariant est guidé par la pré-condition, la boucle et surtout la post-condition. Il existe cependant des algorithmes pour calculer automatiquement des invariants de boucle.

8 – Montrer que pour l'algorithme précédent, la propriété I : $somme \geq 0 \wedge i \geq 0$ est un invariant de boucle. Cet invariant permet-il de montrer que l'algorithme retourne bien la somme des n premiers entiers ? Que permet-il de montrer sur la valeur de *somme* en sortie ?

9 – Montrer que dans l'algorithme suivant, la propriété I : $A = B * Q + R \wedge R \geq 0$ est un invariant de boucle. En déduire que *si il termine*, l'algorithme retourne le quotient et le reste de la division euclidienne de A par B. Montrer que l'algorithme termine en justifiant une fonction de rang. Où intervient la précondition B *entier naturel strictement positif* ?

Algorithme 10 :

Entrées : A un entier naturel et B un entier naturel strictement positif.
Sorties : deux entiers naturels Q et R, quotient et reste de la division euclidienne de A par B
R : *entier* $\leftarrow A$;
Q : *entier* $\leftarrow 0$;
(*) **tant que** $B \leq R$ **faire**
  $R \leftarrow R - B$;
  $Q \leftarrow Q + 1$;
fin
retourner (Q, R)

III D'autres exercices

10 – Considérer l'algorithme suivant. Exécuter avec différentes entrées, par exemple $n = 3$ et $n = 5$. Montrer que $I : X \geq 1$ est un invariant de boucle. Montrer que $Y = (X+1)*(X+2)*\ldots*(n-1)*n$, que nous écrivons $Y = \Pi_{X+1 \leq i \leq n} i$ (ici, le symbole Π désigne le produit, de la même façon que le symbole Σ désigne la somme) est un invariant de boucle. Exprimer puis montrer la correction partielle. Montrer la correction totale. Est-il possible d'affaiblir la précondition, par exemple en imposant n entier naturel tout en préservant la correction totale ? Même question, mais en s'autorisant la modification du test de la boucle.

Algorithme 11 :

Entrées : un entier n, avec $n \geq 1$
Sorties : $n!$, la factorielle de n
Y : *entier* $\leftarrow 1$;
X : *entier* $\leftarrow n$;
(*) **tant que** $X \neq 1$ **faire**
  $Y \leftarrow Y * X$;
  $X \leftarrow X - 1$;
fin
retourner Y

11 – Déterminer puis prouver un invariant sur la plage des valeurs de la variable i au point de programme (*). Justifier que tous les accès au tableau L ont un sens. Déterminer et valider une fonction de rang. Déterminer puis prouver un invariant qui permettra de montrer la correction partielle.

Algorithme 12 :

Entrées : un tableau d'entiers $L[1 \ldots n]$, avec $n \geq 0$
Sorties : un entier, la somme des éléments de L
s : *entier* $\leftarrow 0$;
i : *entier* $\leftarrow 1$;
(*) **tant que** $i \leq n$ **faire**
  $s \leftarrow s + L[i]$;
  $i \leftarrow i + 1$;
fin
retourner s

12 – On considère l'algorithme ci-dessous. Montrer la correction totale et justifier que tous les accès au tableau L ont un sens. Quelle hypothèse doit-on faire sur l'ordre d'évaluation de la conjonction de tests ? Peut-on *relâcher la précondition* en posant comme précondition : *un tableau* $L[1 \ldots n]$ *avec* $n \geq 0$ *et un élément* a tout en conservant les résultats précédents ?

Algorithme 13 : Recherche d'un élément dans un tableau

Entrées : un tableau non-vide d'entiers $L[1 \ldots n]$ et un entier a
Sorties : un booléen, $Vrai$ ssi $a \in L$
i : *entier* $\leftarrow 1$;
(*) **tant que** $i \leq n$ *et* $L[i] \neq a$ **faire**
  $i \leftarrow i + 1$;
fin
retourner $i \leq n$

13 – On considère l'algorithme ci-dessous. Pour chaque post-condition ci-dessous, montrer la correction partielle en justifiant un invariant de boucle à determiner en fonction de chaque post-condition.

1. $m \in L$
2. m : *entier*
3. $m + 0 = m$
4. $\forall i \in [1, n], m \leq L[i]$
5. $m = min\{L\}$

Choisir la post-condition la plus *précise*. Justifier une fonction de rang pour la terminaison de ce programme. Justifier que tous les accès au tableau L ont un sens. Peut-on relâcher la précondition en imposant *un tableau* $L[1 \dots n]$ *avec* $n \geq 0$ tout en conservant les résultats précédents ?

Algorithme 14 :

Entrées : un tableau d'entiers $L[1 \dots n]$, avec $n \geq 1$
Sorties : ...
i : *entier* $\leftarrow 2$;
m : *entier*;
$m \leftarrow L[1]$;
(*) **tant que** $i \leq n$ **faire**
  si $L[i] < m$ **alors**
    $m \leftarrow L[i]$
  fin
  $i \leftarrow i + 1$;
fin
retourner m

14 – Montrer la correction totale de l'algorithme suivant. *Indication :* pour conjecturer un invariant *utile et non-trivial*, tabuler les valeurs de A, N, R en début de boucle pour différentes entrées, par exemple $(a, n) = (2, 8)$ et $(a, n) = (2, 10)$ puis examiner les liens existant entre A, N, R, a, n.

Algorithme 15 :

Entrées : a et n deux entiers naturels
Sorties : l'entier a^n
A : *entier* $\leftarrow a$;
N : *entier* $\leftarrow n$;
R : *entier* $\leftarrow 1$;
(*) **tant que** $N > 0$ **faire**
  si N *pair* **alors**
    $A \leftarrow A \times A$;
    $N \leftarrow N/2$;
  sinon
    $R \leftarrow R \times A$;
    $N \leftarrow N - 1$;
  fin
fin
retourner R