Programmes Corrects par Construction
Partie 1 : Théorie

Pierre-Edouard Portier

1 Introduction

La compétence visée par ce module est l'écriture de programmes corrects par construction.

Pour l'atteindre, il faut acquérir les sous-compétences suivantes :

— Transformer une spécification en langue naturelle en une spécification formelle en logique des prédicats.

— Dériver un programme correct à partir de sa spécification.

Il s'agit de gérer intelligemment la complexité : vérifier la correction d'un programme déjà construit est difficile tandis que dériver un programme correct par construction répartit la complexité en une séquence de décisions maîtrisables. Il s'agira dans un premier temps d'être capable de dériver des programmes séquentiels, puis dans un second temps des programmes concurrents corrects par construction.

Avant de commencer, voici quelques références bibliographiques :

— Backhouse, 2002, *Program Construction the Correct Way*

— Cohen, 1990, Programming in the 1990s an Introduction to the Calculation of Programs

— Dijkstra, 1976, *A Discipline of Programming*

— Feijen, W. H. J., and A. J. M. van Gasteren. *On a Method of Multiprogramming.* Springer Science & Business Media, 1999.

— Gries, 1981, *The Science of Programming*

— Gries, 1994, *A logical approach to discrete math*

— Hehner, 2012, *A practical theory of programming*

— Kaldewaij, 1990, *Programming the Derivation of Algorithms*

— Kourie, Watson, 2012, *The Correctness by Construction Approach to Programming*

— Snepscheut, 1993, *What computing is all about*

— Xue, 1997, *A unified approach for developing efficient algorithmic programs*

2 Spécification

2.1 Triplet de Hoare

`{Q} S {R}` est une expression booléenne appelée **triplet de Hoare**, avec `S` un **programme**, `Q` une **précondition**, et `R` une **postcondition**. `Q` et `R` sont des prédicats (des expressions booléennes). Cette notation signifie que l'exécution du programme `S` à partir d'un état vérifiant `Q` se termine et laisse le système dans un état vérifiant `R`. Préconditions et postconditions sont des prédicats qui décrivent un **ensemble** d'états. La précondition `Q` et la postcondition `R` forment une **spécification** pour le programme `S`.

Soit la spécification suivante :

```
{Q} S {x=y}
```

Pour établir la postcondition `x=y`, le programme `S` doit-il modifier la variable `x`, ou la variable `y` ou les deux variables `x` et `y` ?

`x: E` est la notation utilisée pour représenter une équation `E` d'inconnue `x`. La spécification précédente peut être précisée :

```
(a) {Q} S {x: x=y}
(b) {Q} S {y: x=y}
(b) {Q} S {x,y: x=y}
```

Pour plus de concision, on note également :

```
{Q} x: x=y
```

...la spécification d'un programme qui débute dans un état pour lequel `Q` est vrai, puis établit `x=y` en ne modifiant que la variable `x`. Ici, `y` est une constante.

Le type des variables peut être précisé avec la notation suivante.

```
 var x,y : int {Q}
;x: x=y
```

```
 var x,y : bool {Q}
;x: x ≡ y
```

Une spécification peut être non déterministe. C'est-à-dire que pour un état initial donné, plusieurs états finaux respectent la spécification. Ainsi en est-il de l'exemple suivant :

```
 var x : int {true}
;x: x² = 25
```

Pour déclarer des constantes, on utilise la notation suivante.

```
 cst Y : int
;var x : int {Q}
;x: x=Y
```

Par convention, une lettre majuscule dans une précondition correspond à la déclaration d'une constante.

2.2 Tableaux

Un tableau b est une fonction. Son domaine est un intervalle de l'ensemble des entiers :

$$dom.b = \{i :: b.inf \leq i \leq b.sup\}$$

Son co-domaine dépend du type des objets du tableau.

Pour b une fonction, i une expression entière, e une expression du type du co-domaine de b :

$$(b; i : e).j = \begin{cases} e, & \text{si } i = j. \\ b.j, & \text{sinon.} \end{cases}$$

Ainsi, `b.i := e` $\triangleq$ `b := (b;i:e)`.

La notation suivante est utilisée Pour déclarer un tableau `b` de `N` entiers.

```
var b : array[0..N-1] of int
```

2.3 Quelques spécifications

Calculer une approximation entière de la racine carrée d'un entier donné.

```
 var x : int {0 ≤ N}
;x² ≤ N < (x+1)²
```

`z` doit être égale au produit des entiers naturels `A` et `B`.

```
 cst A,B : int {A ≥ 0 ∧ B ≥ 0}
;var z : int
;z: z = A*B
```

Échanger les valeurs initiales des entiers `x` et `y`.

```
 var x,y : int {x=X ∧ y=Y}
;x,y: x = Y ∧ y = X
```

`q` et `r` doivent être le quotient et le reste de la division entière de $x \geq 0$ par $y > 0$.

```
 var x : int {x ≥ 0}
;var y : int {y > 0}
;var q,r : int
;q,r: q * y + r = x   ∧  0 ≤ r  ∧  r < y
```

Étant donnés l'entier `x` et le tableau d'entiers `b`, le booléen `p` doit valoir : "x est un élément de b".

```
 var b : array[0:N-1] of int
;var x : int
;var p : bool
;p: p ≡ (∃i : 0 ≤ i < N : b.i = x)
```

Trier le tableau d'entiers `b` par ordre croissant.

```
 var b : array[0..N-1] of int {b = B}
;b: perm.b.B   ∧
    (∀i : 0 ≤ i ≤ N-2 : b.i ≤ b.(i+1))
```

Trouver la taille d'un plus long segment nul d'un tableau.

```
 var x : array[0..N-1] of int {N≥0}
;var r : int
;r: (↑p,q : 0 ≤ p ≤ q ≤ N  ∧
            (∀i : p ≤ i < q : x.i = 0) : q-p)
```

Le précis de logique de Gries et Schneider donne des règles utiles pour la manipulation des quantificateurs généralisés.

3 Précondition la plus faible

Soit `Q` une expression booléenne. On note `[Q]` la quantification universelle sur le domaine de définition de `Q` : `(∀t::Q.t)`.

Par la loi de la généralisation de la logique des prédicats, une dérivation de `Q.t` établit `(∀t::Q.t)` si t n'est pas une variable libre de `Q`. C'est pourquoi l'introduction de cette notation pour la quantification universelle est utile : le plus souvent, il suffit de prouver "à l'intérieur des crochets". C'est également la raison pour laquelle, par abus de notation, les crochets sont parfois omis.

Le prédicat `S` est dit plus fort que le prédicat `W` quand : `[S ⇒ W]`.

Par exemple, `(x>5)` est plus fort que `(x>0)` : `[(x>5) ⇒ (x>0)]`. Il faut être sensible aux instantiations telles que `x := 3` sur l'exemple précédent.

`Q[x\a]` dénote la substitution dans l'expression `Q` de chaque occurrence de la variable `x` par le symbole `a`.

```
((x>5) ⇒ (x>0))[x\3] ≡ ⊤
```

`Q[x,y\E,F]` représente l'expression `Q` dans laquelle le symbole `x` est remplacé par l'expression `E` et le symbole `y` est remplacé par l'expression `F`.

Remarque : `Q[x,y\E,F]` $\neq$ `(Q[x\E])[y\F]`.

Il y a une relation directe entre la force des prédicats et la nature des états décrits par ces prédicats. $\mathcal{P}$ dénote l'ensemble de tous les prédicats.

```
(∀ Q : P : [⊥ ⇒ Q ⇒ ⊤])
```

Si `Q'` ⇒ `Q` et `R` ⇒ `R'`, alors :

```
{Q}S{R} ⇒ {Q'}S{R'}
```

Autrement dit, il est toujours possible de renforcer une précondition et d'affaiblir une postcondition. Ce qui amène à imaginer les questions suivantes :

— Étant donnés un programme `S` et une postcondition `R`, quelle est **la plus faible précondition** `wp(S,R)` qui satisfait `{wp(S,R)} S {R}` ?

— Étant donnés un programme `S` et une précondition `Q`, quelle est **la plus forte postcondition** `R` qui satisfait `{Q} S {R}` ?

```
(∀ Q : P : ({Q} S {R}) ⇒ [Q ⇒ wp(S,R)])
```

L'opérateur infixe "." (point) dénote l'application fonctionnelle. Il a la précédence la plus forte et il est associatif à gauche : `wp(S,R)` ≡ `wp.S.R` et `wp.S.R` se lit `(wp.S).R`.

— `{wp.S.R} S {R}` ≡ ⊤

— `wp.S` définit précisément la sémantique du programme S. `wp.S` est la **transformation de prédicat** (predicate transformer) associée au programme `S`.

Grâce à `wp`, le triplet de Hoare peut être défini formellement :

```
{Q} S {R}   ≜   Q ⇒ wp.S.R
```

3.1 Axiomes et propriétés pour wp

`wp.S` suit deux axiomes :

```
wp.S.⊥ ≡ ⊥ (loi du miracle exclus)
wp.S.(X ∧ Y) ≡ wp.S.X ∧ wp.S.Y (distributivité de la conjonction)
```

Au sujet de la loi du "miracle exclus" : ce serait en effet miraculeux s'il existait un état à partir duquel l'exécution d'un programme *se terminait* dans un non-état...

La loi de Leibniz, ci-dessous, est utilisée dans la preuve du prochain théorème :

```
(P ≡ Q) ⇒ (f.P ≡ f.Q)
```

`wp` est **monotone** (i.e. wp préserve l'implication) :

```
(X ⇒ Y) ⇒ (wp.S.X ⇒ wp.S.Y)

preuve :

    wp.S.X ⇒ wp.S.Y
= {P ⇒ Q ≡ P ∧ Q ≡ P}
    wp.S.X ∧ wp.S.Y ≡ wp.S.X
= {conjonctivité}
    wp.S.(X ∧ Y) ≡ wp.S.X
⇐ {leibniz}
    X ∧ Y ≡ X
= {P ⇒ Q ≡ P ∧ Q ≡ P}
    X ⇒ Y
```

La monotonicité permet de prouver un théorème utile :

```
th. affaiblissement de la postcondition
{Q}S{R} ⇐ {Q}S{A} ∧ (A ⇒ R)

preuve :

    {Q}S{A} ∧ (A ⇒ R)
= {déf. triplet de Hoare}
    (Q ⇒ wp.S.A) ∧ (A ⇒ R)
⇒ {monotonicité de wp.S}
    (Q ⇒ wp.S.A) ∧ (wp.S.A ⇒ wp.S.R)
⇒ {transitivité de ⇒}
    Q ⇒ wp.S.R
= {déf. triplet de Hoare}
    {Q}S{R}
```

On prouverait de même une propriété de renforcement de la pré-condition.

La distributivité de la conjonction est posée comme axiome, mais qu'en est-il pour la disjonction ?

```
    wp.S.Q ∨ wp.S.R ⇒ wp.S.(Q∨R)
= {(p ⇒ r) ∧ (q ⇒ r) ≡ (p ∨ q ⇒ r)}
    wp.S.Q ⇒ wp.S.(Q∨R) ∧ wp.S.R ⇒ wp.S.(Q∨R)
⇐ {monotonicité de wp}
    Q ⇒ Q∨R ∧ R ⇒ Q∨R
= {affaiblissement p ⇒ p∨q}
    ⊤
```

Pour des programmes non-déterministes : partant d'un état initial `e ∈ wp.S.Q`, différentes exécutions du programme `S` peuvent terminer en différents états `e' ∈ Q`.

Pour un processus de `jet` d'une pièce : `wp.jet.pile = ⊥` et `wp.jet.face = ⊥`. Mais `wp.jet.(pile ∨ face) = ⊤`.

Pour un programme déterministe : `wp.S.Q ∨ wp.S.R = wp.S.(Q ∨ R)`.

4 Guarded Command Language (GCL)

4.1 skip

```
wp.skip.R = R
```

Par définition des triplets de Hoare, nous avons :

```
{Q}skip{R} ≡ Q ⇒ R
```

La plus simple implémentation pour skip est... de ne rien faire.

4.2 abort

```
wp.abort.R ≡ ⊥
```

Ce qui se traduit en terme de triplets de Hoare par :

```
{Q}abort{R} ≡ (Q ≡ \bot)
```

4.3 composition

```
wp.(S;T).R ≡ wp.S.(wp.T.R)
```

Un lemme utile :

```
{Q}S;T{R} ⇐ {Q}S{H} ∧ {H}T{R}
```

```
preuve :

    {Q} S {H} ∧ {H} T {R}
= {déf. triplet de Hoare}
    (Q ⇒ wp.S.H) ∧ (H ⇒ wp.T.R)
⇒ {monotonicité de wp.S}
    (Q ⇒ wp.S.H) ∧ (wp.S.H ⇒ wp.S.(wp.T.R))
⇒ {transitivité de ⇒}
    Q ⇒ wp.S.(wp.T.R)
= {déf. de la composition}
    Q ⇒ wp.(S;T).R
= {déf. des triplets de Hoare}
    {Q} S;T {R}
```

4.4 affectation

```
wp.(x := E).R = R[x\E]
```

Par exemple :

```
   wp(x := x+1, x>5)
= {déf. affectation}
   (x>5)[x\x+1]
= {déf. substitution}
   x+1>5
= {arithmétique}
   x>4
```

En particulier :

```
(a) {Q[x\E]} x := E {Q}
(b) {Q} x := E {R}
  ≡ {déf. wp}
    Q ⇒ R[x\E]
```

Par exemplei, le programme `x := x+1` implémente-t-il la spécification ci-après ?

```
 var x : int {x>0}
;x: x>1
```

Il s'agit de prouver : `x>0 ⇒ (x>1)[x\x+1]` La preuve part du conséquent sous hypothèse de validité de l'antécédent :

```
   (x>1)[x\x+1]
= {déf. substitution}
   x+1 > 1
= {arithmétique}
   x > 0
= {hyp. antécédent}
   ⊤
```

Autre exemple. Soit le prédicat :

```
var b(0..N-1) : array of int
P : 0 ≤ i ≤ n ∧
    x = (Σ k : 0 ≤ k < i : b.k)
```

`x` est la somme des `i` premiers éléments du tableau `b`. Il s'agit de montrer que le programme :

```
x,i := x+b.i, i+1
```

implémente la spécification :

```
{P ∧ i≠n ∧ i=I}
x,i: P ∧ i = I+1
```

Il faut prouver que :

```
  P ∧ i≠n ∧ i=I
⇒
  (P ∧ i=I+1)[x,i\x+b.i,i+1]
```

La preuve partdu conséquent sous hypothèse de l'antécédent :

```
   0≤i+1≤n ∧
   x+b.i = (Σ k : 0≤k<i+1 : b.k) ∧
   i+1 = I+1
= { i=I et
    0≤i≤n ∧ i≠n
  ≡
    0≤i<n
  ⇒
    0≤i+1≤n }
   x+b.i = (Σ k : 0≤k<i+1 : b.k)
= {split}
   x+b.i = (Σ k : 0≤k<i : b.k) + b.i
= {P}
   vrai
```

Au lieu de vérifier la correction d'une affectation, il est possible de **calculer sa forme correcte**. Par exemple, pour maintenir le prédicat P1 :

```
P1 : x = (Σ k : 0≤k<i : b.k)
```

en utilisant un programme de la forme :

```
i,x := i+1,E
```

Il faut assurer la spécification :

```
{P1} i,x := i+1,E {P1}
```

Pour que cette spécification soit vérifiée, il suffit que :

```
P1 ⇒ P1[i,x\i+1,E]
```

La preuve part du conséquent sous hypothèse de l'antécédent :

```
    P1[i,x\i+1,E]
= {déf. P1, substitution}
    E = (Σ k : 0≤k<i+1 : b.k)
= {split}
    E = (Σ k : 0≤k<i : b.k) + b.i
= {P1}
    E = x + b.i
```

La sémantique de l'affectation multiple suit naturellement de la définition de la substitution multiple.

```
    {x=A ∧ y=B} x,y := y,x {x=B ∧ y=A}
= {déf. de :=}
    (x=A ∧ y=B) ⇒ (x=B ∧ y=A)[x,y\y,x]
= {substitution}
    (x=A ∧ y=B) ⇒ (y=B ∧ x=A)
= {logique}
    ⊤
```

4.5 Alternative

Notation :

```
if B.0      -> S.0
 | B.1      -> S.1
 ...
 | B.(n-1)  -> S.(n-1)
fi
```

Définition :

```
  wp.IF.R
≡
  BB ∧ (∀ i : 0 ≤ i < n : B.i ⇒ wp.(S.i).R)
```

avec :

```
  BB ≡ (∃ i : 0 ≤ i < n : B.i)
```

```
Th. IF

  {Q} IF {R}
≡
  (Q ⇒ BB) ∧
  (∀ i : 0 ≤ i < n : {Q ∧ B.i}S.i{R})

preuve :

Pour simplifier l'écriture, le domaine du quantificateur universel est
omis.

   (Q ⇒ BB) ∧ (∀ i :: {Q ∧ B.i}S.i{R})
= {déf. triplet de Hoare}
   (Q ⇒ BB) ∧ (∀ i :: Q ∧ B.i ⇒ wp.(S.i).R)
= {Afin de pouvoir ensuite sortir Q de la quantification,
          X ∧ Y ⇒ Z ≡ X ⇒ (Y ⇒ Z)}
   (Q ⇒ BB) ∧ (∀ i :: Q ⇒ (B.i ⇒ wp.(S.i).R))
= {Q⇒ distribue sur ∀}
   (Q ⇒ BB) ∧ (Q ⇒ (∀ i :: B.i ⇒ wp.(S.i).R))
= {(X ⇒ Y) ∧ (X ⇒ Z) ≡ X ⇒ Y ∧ Z}
   Q ⇒ (BB ∧ (∀ i :: B.i ⇒ wp.(S.i).R))
= {déf. IF}
   Q ⇒ wp.IF.R
= {déf. des triplets de Hoare}
   {Q} IF {R}
```

Les B.i sont des expressions booléennes appelées "clauses de garde". Si aucune des clauses de garde n'est vraie, l'exécution du programme échoue. Si au moins une des gardes est vraie, alors une des gardes vraie est choisie et les instructions correspondantes sont exécutées. Ainsi, le comportement de l'instruction conditionnelle est potentiellement non déterministe. Ainsi en est-il du programme suivant :

```
if x≤y -> z := y
  |y≤x -> z := x
fi
```

```
¬BB ⇒ IF = abort

preuve :

    IF = abort
= {déf. wp}
    wp.IF.R ≡ wp.abort.R
= {déf. abort}
    wp.IF.R ≡ faux
= {P ≡ faux ≡ ¬P}
    ¬wp.IF.R
= {déf. IF}
    ¬(BB ∧ (∀ i : 0 ≤ i < n : B.i ⇒ wp.(S.i).R))
= {De Morgan}
    ¬BB ∨ ¬(∀ i : 0 ≤ i < n : B.i ⇒ wp.(S.i).R)
⇐ {P ∨ Q ⇐ P}
    ¬BB
```

4.6 Répétition

```
do B.0      -> S.0
  |B.1      -> S.1
  ...
  |B.(n-1)  -> S.(n-1)
od
```

Soit le cas particulier :

```
do B -> S od
```

Théorème de l'invariance :

```
  {Q} do B -> S od {R}
⇐ {Th. de l'invariance}
  Q ⇒ P            ∧ (P est initialement établi.)
  P ∧ B ⇒ wp.S.P   ∧ (P est un invariant.)
  P ∧¬B ⇒ R        ∧ (Postcondition en sortie de boucle.)
  P ∧ B ⇒ t>0      ∧ (Si une itération est possible, alors  t > 0)
  {P∧B} t1 := t ; S {t<t1} (À chaque itération, t diminue.)

Avec t une fonction entière dite "fonction de progrès" (FdP)
```

Dans le cas général :

```
do BB ->
   if B.0      -> S.0
     |B.1      -> S.1
     ...
     |B.(n-1)  -> S.(n-1)
   fi
od

BB ≡ (∃ i : 0≤i<n : B.i)
```

Voici trois stratégies utiles pour construire la boucle au cœur d'un programme :
— `(s-inv-0)` Trouver une première approximation de l'invariant en généralisant la postcondition `R`.
— `(s-inv-1)` Déterminer les conditions sous lesquelles une diminution de la fonction de progrès `t` falsifie l'invariant et modifier `S` pour l'éviter.
— `(s-inv-2)` Déterminer quelles informations supplémentaires peuvent faciliter l'application de (s-inv-1) et introduire de nouvelles variables pour représenter ces informations, modifiant ainsi `P`.

5 Exemple

Trouver un segment de somme minimale dans un tableau non vide.

```
S.i.j ≜ (Σk : i≤k≤j : b.k)

R: s = (↓i,j : 0≤i≤j<N : S.i.j)
```

Application de `(s-inv-0)`. Il faut voir tous les éléments du tableau. En remplaçant la constante `N` par une variable, ils sont vus dans l'ordre croissant de leurs indices. Quelle est la stratégie pour parcourir le tableau dans l'ordre décroissant des indices de ses éléments ?

```
P0: 1≤n≤N
P1: s = (↓i,j : 0≤i≤j<n : S.i.j)
 P: P0 ∧ P1


 n,s:= 1,b.0 {inv P, FdP (N-n)}
;do n ≠ N -> {P ∧ n ≠ N}... {?P[n\n+1]} n:= n+1 {P} od
```

Application de `(s-inv-1)`. Suffit-il de ne rien faire pour établir `P[n\n+1]` en précondition de l'incrément de `n` ? Autrement dit, la propriété ci-dessous est-elle établie ?
`P ∧ n ≠ N ⇒ wp.(n:=n+1).P`
Avec `wp.(n:=n+1).P ≡ P[n\n+1]`.
Remarque : `s'` dénote la valeur mise à jour de `s`. Cette mise à jour permet à l'invariant d'être maintenu au prochain passage de la boucle.

```
    P[n\n+1]
=
    1≤n+1≤N ∧ s' = (↓i,j : 0≤i≤j<n+1 : S.i.j)
= {split pour j=n}
    1≤n+1≤N ∧
    s' = (↓i,j : 0≤i≤j<n : S.i.j) ↓ (↓i : 0≤i≤n : S.i.n)
```

Le premier conjoint est impliqué par `P ∧ n ≠ N`, mais pas le second. L'invariant `P` n'est pas maintenu quand :

```
    (↓i : 0≤i≤n : S.i.n) < (↓i,j : 0≤i≤j<n : S.i.j)
= {P}
    (↓i : 0≤i≤n : S.i.n) < s
```

Comment calculer `(↓i : 0≤i≤n : S.i.n)` ?
Pour ne pas faire un calcul en temps proportionnel à `n`, appliquer `(s-inv-2)` et ajouter une variable à l'invariant :

```
P0:  1≤n≤N
P1:  s = (↓i,j : 0≤i≤j<n : S.i.j)
P2:  c = (↓i : 0≤i < n : S.i.(n-1))
 P:  P0 ∧ P1 ∧ P2
```

Pour découvrir comment mettre à jour `c` pour maintenir la loi `P2` au prochain passage de la boucle, il faut calculer `P2[n\n+1]`.

```
   P2[n\n+1]
=
   c' = (↓i : 0≤i<n+1 : S.i.n)
= {split pour i=n}
   c' = (↓i : 0≤i<n : S.i.n) ↓ S.n.n
=
   c' = (↓i : 0≤i<n : S.i.(n-1) + b.n) ↓ b.n
=
   c' = (↓i : 0≤i<n : S.i.(n-1)) + b.n ↓ b.n
```

Finalement, le programme ci-dessous trouve un segment de somme minimale dans un tableau non vide :

```
 n, s, c := 1, b.0, b.0
;do n ≠ N ->
  c := (c + b.n) ↓ b.n
 ;s := s↓c
 ;n := n+1
 od
```