# Programmes Corrects par Construction
# (1) Théorie

# 1 Introduction

La compétence visée par ce module est la mise en œuvre d'un processus d'assurance et de contrôle qualité à travers 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. Il faut bien noter que préconditions et postconditions sont des prédicats qui décrivent un **ensemble** d'états. La précondition Q et la postcondition R sont 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 bien les deux variables x et y ?

En général, on notera x: E une équation E d'inconnue x. La spécification précédente pourra 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, nous écrirons également :

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

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

Nous pourrons contraindre le type des variables. Par exemple :

```
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
```

Nous pouvons également déclarer des constantes :

```
 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, $b.i := \text{e} \triangleq \text{b} := (b; i : e)$.

## 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)
```

# 3 Précondition la plus faible

Soit Q une expression booléenne. On notera [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".

Nous dirons que le prédicat S est 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.

Nous notons Q[x\a] la substitution dans l'expression Q de chaque occurence de la variable x par le symbole a. Nous introduisons également la substitution multiple Q[x,y\E,F] qui signifie l'expression Q dans laquelle le symbole x est remplacé par l'expression E et le symbole y est remplacé par l'expression F. Il faut bien noter que Q[x,y\E,F] ≠ (Q[x\E])[y\F].

```
(a)  ((x>5) ⇒ (x>0))[x\3] ≡ ⊤
(b)  (∀ X :: (x>5) ⇒ (x>0))[x\X]) ≡ ⊤
```

Il y a une relation directe entre la force des prédicats et la nature des états décrits par ces prédicats. Nous notons $Etats_Q$ les états vérifiant le prédicat $Q$, et $2^P$ l'ensemble de tous les prédicats. Nous avons :

```
(∀ Q : 2^P : [⊥ ⇒ Q ⇒ ⊤])
```

Si Q' ⇒ Q et R ⇒ R', nous avons :

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

Nous noterons l'application fonctionnelle au moyen d'un opérateur infixe noté "." (point) qui a la précédence la plus forte et qui est associatif à gauche : wp(S,R) ≡ wp.S.R et wp.S.R se lit (wp.S).R.

Nous avons :

- {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.

Nous pouvons maintenant définir formellement le triplet de Hoare en tant qu'expression booléenne :

$$\{Q\}\ S\ \{R\} \quad \triangleq \quad Q \Rightarrow wp.S.R$$

## 3.1 Axiomes et propriétés pour wp

wp.S suit deux axiomes :

$$wp.S.\bot \equiv \bot \text{ (loi du miracle exclus)}$$
$$wp.S.(X \wedge Y) \equiv wp.S.X \wedge wp.S.Y \text{ (distributivité de la conjonction)}$$

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 pouvait *se terminer* dans un non-état...

Pour prouver le prochain théorème, nous aurons besoin de **la loi de Leibniz** :

$$(P \equiv Q) \Rightarrow (f.P \equiv f.Q)$$

Nous prouvons la **monotonicité de wp** (wp préserve l'implication) :

$$(X \Rightarrow Y) \Rightarrow (wp.S.X \Rightarrow wp.S.Y)$$

preuve :

$$\begin{array}{ll} & wp.S.X \Rightarrow wp.S.Y \\ = & \{P \Rightarrow Q \equiv P \wedge Q \equiv P\} \\ & wp.S.X \wedge wp.S.Y \equiv wp.S.X \\ = & \{\text{conjonctivité}\} \\ & wp.S.(X \wedge Y) \equiv wp.S.X \\ \Leftarrow & \{\text{leibniz}\} \\ & X \wedge Y \equiv X \\ = & \{P \Rightarrow Q \equiv P \wedge Q \equiv P\} \\ & X \Rightarrow Y \end{array}$$

Grâce à la monotonicité, nous prouvons un théorème utile :

$$\text{th. affaiblissement de la postcondition}$$
$$\{Q\}S\{R\} \Leftarrow \{Q\}S\{A\} \wedge (A \Rightarrow R)$$

preuve :

$$\begin{array}{ll} & \{Q\}S\{A\} \wedge (A \Rightarrow R) \\ = & \{\text{déf. triplet de Hoare}\} \\ & (Q \Rightarrow wp.S.A) \wedge (A \Rightarrow R) \\ \Rightarrow & \{\text{monotonicité de wp.S}\} \\ & (Q \Rightarrow wp.S.A) \wedge (wp.S.A \Rightarrow wp.S.R) \\ \Rightarrow & \{\text{transitivité de } \Rightarrow\} \\ & Q \Rightarrow wp.S.R \\ = & \{\text{déf. triplet de Hoare}\} \\ & \{Q\}S\{R\} \end{array}$$

On prouverait de même une propriété de renforcement de la pré-condition. À vous de le faire en exercice.

Nous posons comme axiome la distributivité de la conjonction, mais qu'en est-il de la disjonction ?

$$
\begin{array}{ll}
 & \mathrm{wp.S.Q} \ \vee \ \mathrm{wp.S.R} \ \Rightarrow \ \mathrm{wp.S.(Q \vee R)} \\
= & \{(p \Rightarrow r) \wedge (q \Rightarrow r) \equiv (p \vee q \Rightarrow r)\} \\
 & \mathrm{wp.S.Q} \ \Rightarrow \ \mathrm{wp.S.(Q \vee R)} \ \wedge \ \mathrm{wp.S.R} \ \Rightarrow \ \mathrm{wp.S.(Q \vee R)} \\
\Leftarrow & \{\text{monotonicité de wp}\} \\
 & \mathrm{Q} \ \Rightarrow \ \mathrm{Q \vee R} \ \wedge \ \mathrm{R} \ \Rightarrow \ \mathrm{Q \vee R} \\
= & \{\text{affaiblissement p} \ \Rightarrow \ \mathrm{p \vee q}\} \\
 & \top
\end{array}
$$

On considère des programmes non-déterministes : partant d'un état initial $e \in wp.S.Q$, différentes exécutions du programme $S$ peuvent terminer en différents états $e' \in Q$.

Pour un processus flip de jet d'une pièce : $wp.flip.pile = \bot$ et $wp.flip.face = \bot$. Mais $wp.flip.(pile \vee face) = \top$.

Pour un programme déterministe : wp.S.Q $\vee$ wp.S.R = wp.S.(Q $\vee$ R).

# 4 Guarded Command Language (GCL)

## 4.1 skip

$$wp.skip.R = R$$

Par définition des triplets de Hoare, nous avons :

$$\{Q\}skip\{R\} \equiv Q \Rightarrow R$$

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

## 4.2 abort

$$wp.abort.R \equiv faux$$

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

$$\{Q\}abort\{R\} \equiv (Q \equiv faux)$$

## 4.3 composition

$$wp.(S;T).R \equiv wp.S.(wp.T.R)$$

Nous avons :

$$\{Q\}S;T\{R\} \Leftarrow \{Q\}S\{H\} \wedge \{H\}T\{R\}$$

preuve :

$$\begin{array}{ll} & \{Q\}\ S\ \{H\} \wedge \{H\}\ T\ \{R\} \\ = & \{\text{déf. triplet de Hoare}\} \\ & (Q \Rightarrow wp.S.H) \wedge (H \Rightarrow wp.T.R) \\ \Rightarrow & \{\text{monotonicité de wp.S}\} \\ & (Q \Rightarrow wp.S.H) \wedge (wp.S.H \Rightarrow wp.S.(wp.T.R)) \\ \Rightarrow & \{\text{transitivité de } \Rightarrow\} \\ & Q \Rightarrow wp.S.(wp.T.R) \\ = & \{\text{déf. de la composition}\} \\ & Q \Rightarrow wp.(S;T).R \\ = & \{\text{déf. des triplets de Hoare}\} \\ & \{Q\}\ S;T\ \{R\} \end{array}$$

## 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
```

Nous avons en particulier :

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

Par exemple : montrer que le programme x := x+1 implémente la spécification :

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

Il s'agit de prouver : x>0 ⇒ (x>1)[x\x+1] Nous partons du conséquent et nous faisons l'hypothèse de la validité de l'antécédent :

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

Autre exemple. Soit le prédicat :

```
var b(0..N−1) : 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]
```

Nous partons du conséquent sous l'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, nous pouvons **calculer sa forme correcte**. Par exemple, supposons qu'il s'agisse de 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
```

Nous avons 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]
```

Nous partons du conséquent sous l'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
```

Grâce à la définition de la substitution multiple, nous pouvons introduire une affectation multiple. Par exemple :

```
    {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

Nous notons IF :

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

Nous définissons IF ainsi :

```
  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 :
```

Nous nous permettons de ne pas écrire le domaine du quantificateuruniversel.

```
    (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 HoareQ 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 (nous le prouverons ci-dessous). 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
```

Considérons le cas particulier :

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

Nous utiliserons en particulier le **théorème d'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
```

Dans le cas général, nous avons :

```
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 qui peuvent être utiles pour construire une 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 borne $t$ falsifie l'invariant et modifier $S$ pour l'éviter.
- (s-inv-2) Déterminer quelles informations supplémentaires faciliteraient l'application de (s-inv-1) et introduire de nouvelles variables pour représenter ces informations, modifiant ainsi $P$.

# 5 Exemple

Il s'agit de trouver un segment de somme minimale dans un tableau non vide.

$S.i.j \triangleq (\Sigma k : i \leq k \leq j : b.k)$

R: $s = (\downarrow i, j : 0 \leq i \leq j < N : S.i.j)$

Application de (s-inv-0). Il faudra visiter chaque élément du tableau. On peut les visiter par ordre croissant de leurs indices en remplaçant la constante $N$ par une variable. (Quelle serait la stratégie pour parcourir le tableau par ordre décroissant des indices de ses éléments ?).

P0: $1 \leq n \leq N$
P1: $s = (\downarrow i, j : 0 \leq i \leq j < n : S.i.j)$
P: P0 $\wedge$ P1

$n, s := 1, b.0$ {inv. P, bf (N–n)}
; do $n \neq N$ –> {P $\wedge$ $n \neq N$} ... {?P[n\n+1]} n:= n+1 {P} od

Application de (s-inv-1). On regarde si P $\wedge$ $n \neq N \Rightarrow$ wp.(n:=n+1).P

wp.(n:=n+1).P
= {gcl}
$1 \leq n+1 \leq N \wedge s = (\downarrow i, j : 0 \leq i \leq j < n+1 : S.i.j)$
= {logique : split pour j=n}
$1 \leq n+1 \leq N \wedge$
$s = (\downarrow i, j : 0 \leq i \leq j < n : S.i.j) \downarrow (\downarrow i : 0 \leq i \leq n : S.i.n)$

Le premier conjoint est impliqué par P $\wedge$ $n \neq N$, mais pas le second. L'invariant P n'est pas maintenu quand :

$(\downarrow i : 0 \leq i \leq n : S.i.n) < (\downarrow i, j : 0 \leq i \leq j < n : S.i.j)$
= {P}
$(\downarrow i : 0 \leq i \leq n : S.i.n) < s$

Comment calculer $(\downarrow i : 0 \leq i \leq n : S.i.n)$ ?
Pour ne pas faire un calcul en temps proportionnel à n, on applique (s-inv-2) et on ajoute une variable à l'invariant :

P0: $1 \leq n \leq N$
P1: $s = (\downarrow i, j : 0 \leq i \leq j < n : S.i.j)$
P2: $c = (\downarrow i : 0 \leq i < n : S.i.(n-1))$
P: P0 $\wedge$ P1 $\wedge$ P2

Pour que cette loi soit maintenue par n:=n+1, il suffit de c := $(c + b.n) \downarrow b.n$
On peut également le découvrir méthodiquement :

```
    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 :

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