Programmes Corrects par Construction
(i) Théorie

1 Spécification

1.1 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^2 = 25
```

1.2 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.

```
 var a,b,z : int {a ≥ 0 ∧ b ≥ 0}
;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 of int
;var x : int
;var p : bool
;p: p ≡ (∃ i : 0 ≤ i < #b : b.i=x)
```

Trier le tableau d'entiers b par ordre croissant.

```
 var b : array of int {bag.b = B}
;b: bag.b = B    ∧
    (∀ i,j : 0 ≤ i ≤ j < #b : b.i ≤ b.j)
```

Trouver la taille du plus long segment du tableau X(0..N−1) qui ne contient que des zéros.

```
 var N : int {N ≥ 0}
;var X(0..N−1) : int
;var r : int
;r: (max p,q : 0 ≤ p ≤ q ≤ N   ∧
                (∀ i : p ≤ i < q : X.i = 0) : q−p)
```

2 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).

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. Par exemple :

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

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 : [faux ⇒ Q ⇒ vrai])
```

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} ≡ vrai
- 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 :

```
{Q} S {R}    ≡    Q ⇒ wp.S.R
```

2.1 Propriétés de wp

Nous postulons que wp.S respecte deux lois :

```
wp.S.faux ≡ faux (excluded miracle)
wp.S.(X ∧ Y) ≡ wp.S.X ∧ wp.S.Y (conjonctivité)
```

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

```
(P ≡ Q) ⇒ (f.P ≡ f.Q)
```

Nous prouvons la **monotonicité de wp** :

```
(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
```

Grâce à la monotonicité, nous prouvons un théorème utile :

```
th. 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}
```

3 Guarded Command Language (GCL)

3.1 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.

3.2 abort

```
wp.abort.R ≡ faux
```

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

```
{Q}abort{R} ≡ (Q ≡ faux)
```

3.3 composition

```
wp.(S;T).R ≡ wp.S.(wp.T.R)
```

Nous avons :

```
{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}
```

3.4 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
```

3.5 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
```

3.6 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** :

```
  {P} do B -> S od {P ∧ ¬B}
⇐
  {P ∧ B} S {P}                  ∧ (invariance)
  {P ∧ B ∧ t=T} S {t<T}          ∧ (progrès)
  P ∧ t≤0 ⇒ ¬B                     (borne)

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)
```

Pour prouver qu'une répétition établit une postcondition R, nous pouvons procéder ainsi :

```
  {P} do B -> S od {P ∧ ¬B} ∧
  P ∧ ¬B ⇒ R
⇒
  {P} do B -> S od {R}
```

Il est souvent nécessaire d'établir l'invariant avant d'entamer la répétition :

```
{Q} S0 {P} ; {P} do B -> S od {R}
```

4 Exemple

Prenons l'exemple de la division entière :

```
R: 0≤r ∧ r<y ∧ q*y+r = x

 var r,q,x,y : int {Q: x≥0 ∧ y>0}
;q,r: R
```

Bien sûr, la solution ne devra pas utiliser l'opération de division.

4.1 Déterminer l'invariant

Nous cherchons P et B tels que : $P \wedge \neg B \Rightarrow R$. Ainsi, l'invariant P peut-être construit par **affaiblissement de la postcondition**. Si R est sous la forme d'une conjonction, l'affaiblissement le plus évident consiste à supprimer un des termes de la conjonction. Le conjoint $q*y+r = x$ lie entre elles toutes les variables. Il a toutes les chances d'appartenir à l'invariant. Nous proposons de supprimer le conjoint $r<y$, dont la négation $r \geq y$ devient la clause de garde B :

```
inv    P: 0≤r ∧ q*y+r = x
garde  B: r≥y
```

D'où l'esquisse de programme :

```
 {Q: 0≤x ∧ 0<y}
 S0
 {inv P: 0≤r ∧ q*y+r = x}
;do r≥y -> S od
 {P ∧ r<y, d'où: R}
```

4.2 Établir l'invariant

Peut-être est-il clair que pour établir l'invariant il suffit de :

```
S0: q,r := 0,x
```

Pour s'en convaincre, calculons cette affectation. La forme de S0 doit être :

```
S0: q,r := E,F
```

Nous avons :

```
    wp.(q,r := E,F).P
= {déf. de P et de l'affectation}
    0 ≤ F    ∧    E*y + F = x
= {choisir E=0 pour éviter toute division}
    0 ≤ F    ∧    F = x
= {choisir F=x pour rendre vrai le second conjoint}
    0 ≤ F
= {Q}
  vrai
```

4.3 Choisir la fonction de borne

L'évolution de la fonction de borne ("bound function", notée bf) doit permettre de falsifier la garde : r≥y. Puisque y est une constante, nous avons :

```
bf t: r
```

4.4 Développer le corps de la boucle

Le corps de la boucle doit :
— diminuer t, avec un programme de la forme : r:=r−k pour k>0 fixé
— maintenir P : {P ∧ r≥y} S {P}
Nous cherchons à calculer :

```
{P ∧ r≥y} r,q := r−k,E {P}
```

```
    wp.(r,q := r−k,E).P
= {déf. P et affectation}
    r≥k ∧ E*y + r−k = x
= {P}
    r≥k ∧ E*y + r−k = q*y + r
= {arithmétique}
    r≥k ∧ E = q + k/y
= {pas d'opération de division,
    donc k doit être un multiple de y
    nous choisissons k=y}
    r≥y ∧ E = q+1
= {garde}
    E = q+1
```

D'où :

```
S: r,q := r−y,q+1
```

4.5 Solution

```
 {Q: 0≤x ∧ 0<y}
 q,r := 0,x
 {inv P: 0≤r ∧ q*y+r=x}
;do r≥y -> r,q := r-y,q+1 od
 {P ∧ r<y, d'où R}
```