Programmes Corrects par Construction

Pierre-Edouard Portier

Version Septembre 2020

1 Introduction

1.1 Compétences

La compétence visée par ce module est l'écriture de programmes corrects par construction. Pour l'atteindre, les sous-compétences suivantes sont requises :

— 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. Nous traitons principalement de la dérivation programmes séquentiels. Nous introduisons également à la construction de programmes concurrents.

1.2 Références

— 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 expressions booléennes (ou prédicats). $\{Q\}S\{R\}$ énonce 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.

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

2.2 Exemples

2.2.1 Produit

Calculer le produit des entiers naturels `A` et `B`.

$\{A \geq 0 \wedge B \geq 0\}$
`z : int`
`;S`
$\{z = A * B\}$

2.2.2 Swap

Échanger les valeurs des variables entières x et y.

`x,y : int` $\{x = X \wedge y = Y\}$
`;S`
$\{x = Y \wedge y = X\}$

2.2.3 Racine carrée entière

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

`x : int`
$\{0 \leq N\}$
`;S`
$\{x^2 \leq N < (x+1)^2\}$

2.2.4 Division entière

q et r doivent être le quotient et le reste de la division entière de X par Y.

$\{X \geq 0 \quad \wedge \quad Y > 0\}$
`q,r : int`
`;S`
$\{q \times Y + r = X \quad \wedge \quad 0 \leq r \quad \wedge \quad r < Y\}$

3 Quantificateurs généralisés

Une spécification peut utiliser des quantificateurs (e.g. $\forall$, $\exists$, Σ, Π...). Il s'agit toujours d'appliquer un opérateur binaire commutatif et associatif aux éléments d'un ensemble. Pour introduire la notation employée pour les quantificateurs, prenons l'exemple de la somme des cubes des trois premiers entiers strictement positifs.

$$\left(\sum k : 1 \leq k \leq 3 : k^3\right)$$

La notation comprend trois parties séparées par deux caractère "deux points".

$$(\star vl : domaine : terme)$$

La première partie est composée du symbole du quantificateur (e.g. Σ pour le quantificateur associé à l'opérateur binaire $+$) et des noms des variables liées (vl). Nous rencontrerons principalement les quantificateurs suivants :

— Σ pour l'opérateur binaire *somme* noté $+$
— Π pour l'opérateur binaire *produit* noté $\times$
— $\exists$ pour l'opérateur binaire booléen *ou* noté $\vee$
— $\forall$ pour l'opérateur binaire booléen *et* noté $\wedge$
— $\uparrow$ pour l'opérateur binaire *max* noté $\uparrow$
— $\downarrow$ pour l'opérateur binaire *min* note $\downarrow$
— $\ldots$
— $\star$ pour l'opérateur binaire $\bigstar$

La seconde partie est une expression booléenne qui définit le domaine des valeurs que peuvent prendre la ou les variables liées. La dernière est un terme qui dépend des variables liées, on peut le voir comme une fonction appliquée à chaque point du domaine avant d'opérer la combinaison par l'opérateur binaire.

3.1 Règles

L'utilisation d'une notation générique permet d'énoncer des règles qui s'appliquent à tout quantificateur.

3.1.1 Domaine vide

Lorsque le domaine est vide (i.e. $dom \equiv \bot$), le quantificateur vaut l'élément neutre de son opérateur binaire associé.

— $(\Sigma \ldots : \bot : \ldots) = 0$
— $(\Pi \ldots : \bot : \ldots) = 1$
— $(\exists \ldots : \bot : \ldots) = \bot$
— $(\forall \ldots : \bot : \ldots) = \top$
— $(\uparrow \ldots : \bot : \ldots) = -\infty$
— $(\downarrow \ldots : \bot : \ldots) = \infty$
— $\ldots$

3.1.2 Singleton

$$(\star k : k = e : t) = t[k \backslash e]$$

3.1.3 Séparation

$$(\star k : P : T) = (\star k : P \wedge Q : T) \bigstar (\star k : P \wedge \neg Q : T)$$

3.1.4 Séparation pour un élément unique du domaine

Cette règle découle des deux précédentes. Il est plus simple de l'expliquer sur un exemple.

$$
\begin{array}{ll}
 & \left(\Sigma k : 0 \leq k \leq N : 2^k\right) \\
= & \{\text{Séparation sur } k = 0 \vee k \neq 0\} \\
 & \left(\Sigma k : 0 \leq k \leq N \wedge k = 0 : 2^k\right) + \left(\Sigma k : 0 \leq k \leq N \wedge k \neq 0 : 2^k\right) \\
= & \{\text{Simplification des domaines, en supposant que } N \geq 0\} \\
 & \left(\Sigma k : k = 0 : 2^k\right) + \left(\Sigma k : 1 \leq k \leq N : 2^k\right) \\
= & \{\text{Règle du singleton et } 2^0 = 1\} \\
 & 1 + \left(\Sigma k : 1 \leq k \leq N : 2^k\right)
\end{array}
$$

À partir de maintenant, on écrira plus simplement :

$$
\begin{array}{ll}
 & \left(\Sigma k : 0 \leq k \leq N : 2^k\right) \\
= & \{\text{Séparation pour } k = 0 \text{ en supposant } N \geq 0\} \\
 & 1 + \left(\Sigma k : 1 \leq k \leq N : 2^k\right)
\end{array}
$$

3.2 Exemples de spécifications avec quantificateurs

3.2.1 Recherche

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

```
 f(i:0 ≤ i < N) : array of int
;x : int
;p : bool
;S
 {p ≡ (∃i : 0 ≤ i < N : f.i = x)}
```

3.2.2 Tri

Trier le tableau d'entiers f par ordre croissant.

```
 f(i:0 ≤ i < N) : array of int {f = F}
;S
 {perm.f.F}
 {(∀i : 0 ≤ i ≤ N − 2 : f.i ≤ f.(i + 1))}
```

3.2.3 Problème de segment

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

```
 f(i:0 ≤ i < N) : array of int {N ≥ 0}
;r : int
;S
 {r = (↑ p, q : 0 ≤ p ≤ q ≤ N ∧ (∀i : p ≤ i < q : f.i = 0) : q − p)}
```

4 Précondition la plus faible

4.1 Force des prédicats

Le prédicat S est dit plus fort que le prédicat W quand, quelles que soient les valeurs que prennent leurs variables libres, $S \Rightarrow W$. Par exemple, $(x > 5)$ est plus fort que $(x > 0)$:

$$(\forall x :: (x > 5) \Rightarrow (x > 0))$$

Que l'on note également, an abréviant par des crochets la quantification universelle sur le domaine de définition des prédicats :

$$[(x > 5) \Rightarrow (x > 0)]$$

Sur cet exemple, il est intéressant de réfléchir à une situation telle que $x = 3$.

4.2 Substitutions

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

$$((x > 5) \Rightarrow (x > 0))\,[x\backslash 3] \equiv \top$$ [1]

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

4.3 Précondition la plus faible

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.

$$(\forall Q : \mathcal{P} : [\bot \Rightarrow Q \Rightarrow \top])$$

Si $Q' \Rightarrow Q$ et $R \Rightarrow R'$, alors : $\{Q\}S\{R\} \;\Rightarrow\; \{Q'\}S\{R'\}$. Autrement dit, il est toujours possible de renforcer une précondition et d'affaiblir une postcondition.

— Étant donnés un programme S et une postcondition R, quelle est **la plus faible précondition** $wp(S, R)$ qui satisfait $\{wp(S, R)\}S\{R\}$?
Autrement dit, $(\forall Q : \mathcal{P} : (\{Q\}S\{R\}) \;\Rightarrow\; [Q \Rightarrow wp(S, R)])$

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

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

— $\{wp.S.R\}S\{R\} \equiv \top$

— $wp.S$ donne un sens formel au programme S.

4.4 Triplets de Hoare et précondition la plus faible

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

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

1. Nous notons les booléens vrai et faux par $\top$ et $\bot$

5 Guarded Command Language (GCL)

On utilise la notion de précondition la plus faible pour donner un sens formel aux instructions d'un langage de programmation.

5.1 skip

$$wp.skip.R = R$$

Par définition des triplets de Hoare, nous avons :

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

L'implémentation la plus simple pour `skip` consiste à ne rien faire.

5.2 abort

$$wp.abort.R \equiv \bot$$

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

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

5.3 composition

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

Un lemme utile :

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

5.4 affectation

$$wp.(x := E).R \equiv R[x\backslash E]$$

Par exemple :

$$\begin{aligned}
& wp(x := x+1, x > 5) \\
=& \{\text{Déf. affectation}\} \\
& (x > 5)[x\backslash x+1] \\
=& \{\text{Déf. substitution}\} \\
& x+1 > 5 \\
=& \{\text{Arithmétique}\} \\
& x > 4
\end{aligned}$$

En particulier :

(a) $\{Q[x\backslash E]\}x := E\{Q\}$

(b) $\{Q\}x := E\{R\} \equiv Q \Rightarrow R[x\backslash E]$

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

$$
\begin{array}{ll}
 & \{x = A \wedge y = B\}x, y := y, x\{x = B \wedge y = A\} \\
= & \{\text{Déf. :=}\} \\
 & (x = A \wedge y = B) \Rightarrow (x = B \wedge y = A)[x, y \backslash y, x] \\
= & \{\text{Substitution}\} \\
 & (x = A \wedge y = B) \Rightarrow (y = B \wedge x = A) \\
= & \{\text{Logique}\} \\
 & \top
\end{array}
$$

5.5 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)
```

Les $B.i$ sont des expressions booléennes appelées "clauses de garde". Si aucune n'est vérifiée, l'exécution du programme échoue. Sinon, l'une des clauses de garde vraies est choisie et l'instruction correspondante est exécutée. Ainsi, le comportement de l'instruction conditionnelle est potentiellement non déterministe :

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

5.6 Répétition

```
  {Q} do B -> S od {R}
⇐ {Théorème 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 ou bound function (bf)

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 post-condition R.
— `(s-inv-1)` Déterminer les conditions sous lesquelles une diminution de la fonction de progrès t falsifie l'invariant et modifier le programme en conséquence.
— `(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.

6 Exemple

Trouver un segment de somme minimale dans un tableau non vide [2].

6.1 Spécification

```
 f(i:0 ≤ i < N) : array of int {N > 0}
;r : int
;S
{r = (↓ i, j : 0 ≤ i ≤ j < N : S.i.j)}

Avec :
S.i.j ≜ (Σk : i ≤ k ≤ j : f.k)
```

6.2 Première approximation d'un invariant

Application de `(s-inv-0)`. Pour résoudre le problème, il semble nécessaire de voir au moins une fois chaque élément du tableau. En affaiblissant la postcondition par le remplacement de la constante `N` par une variable `n`, on obtient un invariant qui conduit à un programme où les éléments sont vus dans l'ordre croissant de leurs indices.

$$\begin{aligned} P0 &: 1 \leq n \leq N \\ P1 &: r = (\downarrow i, j : 0 \leq i \leq j < n : S.i.j) \\ P &: P0 \wedge P1 \end{aligned}$$

En général, à chaque variable son invariant. Ici, $P0$ est la loi qui contrôle la valeur que peut prendre n, tandis que $P1$ contrôle la valeur de r.

Pour établir $P0$ avant d'entrer dans la boucle principale, il suffit d'initialiser n à 1. Il faut également établir $P1$ en initialisant r. Pour ce faire, il suffit de calculer $P1[n\backslash 1]$ car r dépend de

2. Voir l'histoire intéressante de ce type de problèmes : https://en.wikipedia.org/wiki/Maximum_subarray_problem

n qui est initialisée à 1.

$$
\begin{array}{ll}
& P1[n\backslash 1] \\
= & \{\text{Déf. de } P1\} \\
& r = (\downarrow i, j : 0 \leq i \leq j < 1 : S.i.j) \\
= & \{\text{Règle du singleton. Le seul élément du domaine du quantificateur est } (i, j) = (0, 0)\} \\
& r = S.0.0 \\
= & \{\text{Déf. de } S\} \\
& r = (\Sigma k : 0 \leq k \leq 0 : f.k) \\
= & \{\text{Règle du singleton. Le seul élément du domaine du quantificateur est } k = 0\} \\
& r = f.0
\end{array}
$$

L'invariant P est conçu pour que la postcondition soit établie quand $n = N$. Donc, la condition d'entrée dans la boucle est $n \neq N$. Nous proposons comme fonction de progrès (ou *bound function* abrégée bf) $N - n$. Le progrès est assuré en choisissant pour dernière instruction de la boucle : `n:= n+1`. Il reste à compléter la boucle pour que l'invariant P soit maintenu.

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

6.3 Raffinement de l'invariant

Application de `(s-inv-1)`. Suffit-il de ne rien faire pour que `n:= n+1` maintienne P ? Autrement dit, la propriété ci-dessous est-elle établie ?

$$
\begin{array}{ll}
& P \wedge n \neq N \;\Rightarrow\; wp.(n := n + 1).P \\
= & \{\text{Par déf de l'affectation}\} \\
& P \wedge n \neq N \;\Rightarrow\; P[n\backslash n + 1]
\end{array}
$$

Nous essayons de prouver cette implication en partant du conséquent et sous hypothèse de l'antécédent. Remarque : r' dénote la valeur mise à jour de r. Cette mise à jour permet à l'invariant d'être maintenu au prochain passage de la boucle.

$$
\begin{array}{ll}
& P[n\backslash n + 1] \\
= & \{\text{Par déf de } P\} \\
& 1 \leq n + 1 \leq N \;\wedge\; s' = (\downarrow i, j : 0 \leq i \leq j < n + 1 : S.i.j) \\
= & \{\text{Séparation pour } j = n\} \\
& 1 \leq n + 1 \leq N \;\wedge \\
& r' = (\downarrow i, j : 0 \leq i \leq j < n : S.i.j) \downarrow (\downarrow i : 0 \leq i \leq n : S.i.n) \\
\Leftarrow & \{P0, P1 \text{ et } n \neq N\} \\
& r' = r \downarrow (\downarrow i : 0 \leq i \leq n : S.i.n)
\end{array}
$$

Ainsi, l'invariant P n'est pas maintenu quand :

$$(\downarrow i : 0 \leq i \leq n : S.i.n) \;<\; r$$

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

$$
\begin{array}{rcl}
P0 & : & 1 \le n \le N \\
P1 & : & r = (\downarrow\ i,j\ :\ 0 \le i \le j < n\ :\ S.i.j) \\
P2 & : & c = (\downarrow\ i\ :\ 0 \le i < n\ :\ S.i.(n-1)) \\
P & : & P0 \wedge P1 \wedge P2
\end{array}
$$

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

$$
\begin{array}{ll}
& P2[n\backslash n+1] \\
= & \{\text{Déf. de } P2\} \\
& c' = (\downarrow\ i\ :\ 0 \le i < n+1\ :\ S.i.n) \\
= & \{\text{Séparation pour } i = n \text{ et on a bien } n+1 > 0\} \\
& c' = (\downarrow\ i\ :\ 0 \le i < n\ :\ S.i.n) \downarrow S.n.n \\
= & \{S.n.n = f.n \quad \text{et} \quad S.i.n = S.i.(n-1) + f.n\} \\
& c' = (\downarrow\ i\ :\ 0 \le i < n\ :\ S.i.(n-1) + f.n) \downarrow f.n \\
= & \{\text{Par distributivité, la constante } f.n \text{ peut être sortie du quantificateur.}\} \\
& c' = (\downarrow\ i\ :\ 0 \le i < n\ :\ S.i.(n-1)) + f.n \downarrow f.n \\
= & \{\text{Déf. de } P2\} \\
& c' = c + f.n \downarrow f.n
\end{array}
$$

Nous avons calculé comment mettre à jour c pour maintenir $P2$. Il reste à initialiser c pour établir $P2$ avant d'entrer dans la boucle. Pour ce faire, il suffit de calculer $P2[n\backslash 1]$ car c dépend de n qui est initialisée à 1.

$$
\begin{array}{ll}
& P2[n\backslash 1] \\
= & \{\text{Déf. de } P2\} \\
& c = (\downarrow\ i\ :\ 0 \le i < 1\ :\ S.i.0) \\
= & \{\text{Règle du singleton. Le seul élément du domaine du quantificateur est } i = 0\} \\
& c = S.0.0 \\
= & \{\text{Déf. de } S\} \\
& c = (\Sigma k\ :\ 0 \le k \le 0\ :\ f.k) \\
= & \{\text{Règle du singleton. Le seul élément du domaine du quantificateur est } k = 0\} \\
& c = f.0
\end{array}
$$

6.4 Programme final

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