Programmes Corrects par Construction

Pierre-Edouard Portier

Version Septembre 2022

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.

1.2 Quelques 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

Le programme `S` calcule 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

Le programme `S` échange 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

Le programme `S` calcule une approximation entière de la racine carrée d'un entier N.

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

2.2.4 Division entière

Après l'exécution de `S`, `q` et `r` sont le quotient et le reste de la division entière de `X` par `Y`.

```
 {X ≥ 0   ∧   Y > 0}
 q,r : int
;S
 {q × Y + r = X   ∧   0 ≤ r   ∧   r < Y}
```

3 Quantificateurs généralisés

Une spécification peut utiliser des quantificateurs (e.g. *Quelque soit* noté ∀, *Il existe* noté ∃, *Somme* noté Σ, *Produit* noté Π…). Il s'agit toujours d'appliquer un opérateur binaire commutatif et associatif aux éléments d'un ensemble. Pour introduire la notation retenue, 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 caractères "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). Les quantificateurs les plus utilisés sont :

- Σ pour l'opérateur binaire *somme* noté +
- Π pour l'opérateur binaire *produit* noté ×
- ∃ pour l'opérateur binaire booléen *ou* noté ∨
- ∀ pour l'opérateur binaire booléen *et* noté ∧
- ↑ pour l'opérateur binaire *max* noté ↑
- ↓ pour l'opérateur binaire *min* note ↓
- …

Pour que l'explication de la notation reste générique, nous avons utilisé un symbole inventé pour représenter un quantificateur quelconque (viz. ☆, associé à l'opérateur binaire ★).
La seconde partie est une expression booléenne qui définit le domaine des valeurs que peuvent prendre la ou les variables liées. La dernière partie est un terme qui dépend des variables liées, on peut le voir comme une fonction appliquée à chaque point du domaine avant d'en 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. $dom \equiv \bot$ [1]), le quantificateur vaut l'élément neutre de son opérateur binaire associé.

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

1. Nous notons les booléens vrai et faux par les symboles ⊤ et ⊥. Nous notons l'égalité entre valeurs booléennes, aussi appelée équivalence, par le symbole ≡. Remarque : si l'égalité entre valeurs quelconques est avant tout transitive, l'égalité entre valeurs booléennes est également associative.

Nous pouvons bien comprendre la nécessité de cette règle sur un exemple simple.

$(\Sigma k : 1 \leq k \leq 2 : k)$
= {Séparation pour $k = 1$. La séparation sera vue plus bas.
C'est une simple application de la commutativité et de l'associativité.}
$1 + (\Sigma k : 2 \leq k \leq 2 : k)$
= {Séparation pour $k = 2$. Observer que $3 \leq k \leq 2 \equiv \bot$.}
$1 + 2 + (\Sigma k : \bot : k)$
= {Pour que le résultat soit bien $1 + 2$, il faut que la règle du domaine vide s'applique.}
$1 + 2 + 0$

3.1.2 Singleton

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

3.1.3 Séparation

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

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.

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

En pratique, on écrit plus simplement :

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

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 `f`".

```
 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. $perm.X.Y$ signifie que les éléments du tableau X sont une permutation des éléments du tableau Y.

```
 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

Nous appelons *segment* d'un tableau, une séquence d'éléments contigus.
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.
$Q[x, y\backslash E, F]$ 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 :

$\{Q\}S\{R\} \quad \triangleq$ [2] $\quad Q \Rightarrow wp.S.R$

2. $\triangleq$ signifie "est égal par définition"

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 :

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

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 :

$\{x = A \wedge y = B\}x, y := y, x\{x = B \wedge y = A\}$
$=$ {Déf. :=}
$(x = A \wedge y = B) \Rightarrow (x = B \wedge y = A)[x, y \backslash y, x]$
$=$ {Substitution}
$(x = A \wedge y = B) \Rightarrow (y = B \wedge x = A)$
$=$ {Logique}
$\top$

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 clause de garde de l’alternative 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)
```

6 Segment de somme minimale

Trouver un segment de somme minimale dans un tableau non vide. [3]

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

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

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

dans l'ordre croissant de leurs indices.

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

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 n qui est initialisée à 1.

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

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,f.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

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

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

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.

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

Ainsi, il est nécessaire de compléter le programme car l'invariant P n'est pas maintenu quand :

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

Comment calculer $(\downarrow i : 0 \leq i \leq n : S.i.n)$? Pour ne pas faire un calcul en temps proportionnel à n, ajouter une variable à l'invariant :

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

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
```

7 Segment de somme minimale en Dafny

Dafny [4] est un langage de programmation qui vérifie la correction d'assertions formelles ajoutées à un programme. La syntaxe de Dafny est assez proche de celle que nous avons utilisée jusqu'ici.

```
method minsum(a: array<int>) returns (k: int, l: int)
  ensures 0 <= k <= l <= a.Length
  ensures forall i,j :: 0 <= i <= j <= a.Length ==> sum(a,i,j) >= sum(a,k,l)
{
  // TODO
}

function sum(a: array<int>, i:int, j:int): int
  reads a
  requires 0 <= i <= j <= a.Length
  decreases j-i
{
  if i==j then 0 else sum(a, i, j-1) + a[j-1]
}
```

Nous introduisons un invariant classique avec une variable n qui prend le rôle de `a.Length`.

4. https://dafny.org/

```
invariant 0 <= k <= l <= n <= a.Length
invariant r == sum(a, k, l)
invariant forall i,j :: 0 <= i <= j <= n ==> sum(a,i,j) >= r
```

Nous calculons par induction comment faire évoluer les valeurs de r, k et l pour maintenir l'invariant quand `n:=n+1`.

```
  forall i,j :: 0 <= i <= j <= n+1 ==> sum(a,i,j) >= r'
== // séparation pour j == n+1
  r' <= r && forall i :: 0 <= i <= n+1 ==> sum(a,i,n+1) >= r'
== // invariant s = sum(a,m,n)
   // invariant forall i :: 0 <= i <= n ==> sum(a,i,n) >= s
  r' <= r && r' <= s'
==
  if (r <= s') then r' == r else r' == s' && k' == m' && l' == n+1
```

Nous avons introduit un nouvel invariant qu'il faut maintenir.

```
  forall i :: 0 <= i <= n+1 ==> sum(a,i,n+1) >= s'
== // séparation pour i == n+1 ; définition de sum ; sum(n+1,n+1) == 0
  (forall i :: 0 <= i <= n+1 ==> sum(a,i,n) + a[n] >= s')  &&  0 >= s'
== // définition of s
  s' <= s + a[n]  &&  0 >= s'
==
  if (s + a[n] < 0) then s' == s + a[n] else s' == 0 && m' == n+1
```

D'où le programme, vérifié par le moteur d'inférence de Dafny :

```
method minsum(a: array<int>) returns (k: int, l: int)
  ensures 0 <= k <= l <= a.Length
  ensures forall i,j :: 0 <= i <= j <= a.Length ==> sum(a,i,j) >= sum(a,k,l)
{
  k, l := 0, 0;
  var n, r, s, m := 0, 0, 0, 0;
  while (n < a.Length)
    invariant 0 <= k <= l <= n <= a.Length
    invariant r == sum(a, k, l)
    invariant forall i,j :: 0 <= i <= j <= n ==> sum(a,i,j) >= r
    invariant 0 <= m <= n
    invariant s == sum(a, m, n)
    invariant forall i :: 0 <= i <= n ==> sum(a,i,n) >= s
    decreases a.Length - n
  {
    if (s + a[n] < 0)
    {
      s := s + a[n];
    }
    else
    {
      s := 0;
      m := n+1;
    }
    if (s < r)
    {
      r := s;
      k := m;
      l := n+1;
    }
    n := n + 1;
  }
}
```

8 Segment $A - B$ de somme maximale

```
 f(i:0 ≤ i < N) : array of int {N ≥ 1}
;r : int
;seg-A-B-max
```
$\{r = (\uparrow i, j : 0 \le i \le j < N \ \wedge\ f.i = A \ \wedge\ f.j = B : S.i.j)\}$

Avec :
$S.i.j \triangleq (\Sigma k : i \le k \le j : f.k)$

Introduction d'un invariant en remplaçant la constante N par une variable n.

$P0 : 0 \le n \le N$

$P1 : r = (\uparrow i, j : 0 \le i \le j < n \ \wedge\ f.i = A \ \wedge\ f.j = B : S.i.j)$

$P0$ est établi avec : `n := 0`. $P1$ est établi avec : `r := `$-\infty$.
Calcul de la mise à jour de `r` pour maintenir $P1$ après un incrément de `n` :

$P1[n \backslash n+1]$
$=$ {Déf. de $P1$}
$r' = (\uparrow i, j : 0 \le i \le j < n+1 \ \wedge\ f.i = A \ \wedge\ f.j = B : S.i.j)$
$=$ {Séparation pour $j = n$; $P1$}
$r' = r \uparrow (\uparrow i : 0 \le i \le n \ \wedge\ f.i = A \ \wedge\ f.n = B : S.i.n)$
$=$ {$f.n = B$? }

$$r' = \begin{cases} r, & \text{si } f.n \neq B \\ r \uparrow (\uparrow i : 0 \le i \le n \ \wedge\ f.i = A : S.i.n), & \text{si } f.n = B \end{cases}$$

$P2 : s = (\uparrow i : 0 \le i \le n-1 \ \wedge\ f.i = A : S.i.(n-1))$

$P2$ est établi par l'initialisation : `s := `$-\infty$.

$P2[n \backslash n+1]$
$=$ {Déf. de $P2$}
$s' = (\uparrow i : 0 \le i \le n \ \wedge\ f.i = A : S.i.n)$
$=$ {$S.i.n = f.n + S.i.(n-1)$}
$s' = f.n + (\uparrow i : 0 \le i \le n \ \wedge\ f.i = A : S.i.(n-1))$
$=$ {Séparation pour $i = n$; $S.n.(n-1) = 0$; $f.n = A$? ; $P2$}

$$s' = \begin{cases} f.n + s, & \text{si } f.n \neq A \\ f.n + (s \uparrow 0), & \text{si } f.n = A \end{cases}$$

```
 f(i:0 ≤ i < N) : array of int {N ≥ 1}
;n, r, s: int
;n, r, s := 0, -∞, -∞
;do n ≠ N ->
  if f.n ≠ A -> s := f.n + s
    |f.n = A -> s := f.n + (s ↑ 0)
  fi
 ;if f.n ≠ B -> skip
    |f.n = B -> r := r ↑ s
  fi
 ;n := n+1
od
```

9 Segment $A - B$ de somme maximale en Dafny

Partons de la spécification suivante.

```
function sum(a: array<int>, i:int, j:int): int
  reads a
  requires a.Length > 0
  requires 0 <= i <= j < a.Length
  decreases j-i
{
  if i==j then a[i] else sum(a, i, j-1) + a[j]
}

method maxsumab(a: array<int>, A: int, B: int) returns (k: int, l: int)
  requires a.Length > 0
  ensures 0 <= k <= l < a.Length
  ensures forall i,j :: 0 <= i <= j < a.Length &&
                        a[i] == A && a[j] == B
                        ==> sum(a,i,j) <= sum(a,k,l)
```

Nous introduisons un invariant classique avec une variable `n` qui prend le rôle de `a.Length`.

```
invariant 0 <= k <= l < n <= a.Length
invariant r == sum(a, k, l)
invariant forall i,j :: 0 <= i <= j < n && a[i] == A && a[j] == B
                        ==> sum(a, i, j) <= r
```

Cherchons à maintenir l'invariant pour `n:=n+1`.

```
  forall i,j :: 0<=i<=j < n+1 && a[i]==A && a[j]==B ==> sum(a,i,j) <= r'
== // séparation pour j==n ; invariant
  r'>=r && forall i :: 0<=i<=n && a[i]==A && a[n]==B ==> sum(a,i,n) <= r'
==
  if (a[n] != B)
  then r'==r
  else r'>=r && forall i :: 0<=i<=n && a[i]==A ==> sum(a,i,n) <= r'
== // invariant s == sum(a,m,n-1)
   // invariant forall i :: 0<=i<n && a[i]==A ==> sum(a,i,n-1) <= s
  if (a[n] != B)
  then r' == r
  else if (r < s')
       then r' == s' && k' == m' && l' == n
       else r' == r
```

Nous avons introduit un nouvel invariant qu'il faut également maintenir.

```
  forall i :: 0<=i<=n && a[i]==A ==> sum(a,i,n) <= s'
== // séparation pour i==n ; définition de sum
  (forall i :: 0<=i<n && a[i]==A ==> sum(a,i,n-1) + a[n] <= s') &&
  a[n]==A ==> sum(n,n) <= s'
== // invariant ; sum(n,n) == a[n]
  s' >= s + a[n]  &&  a[n] == A ==> a[n] <= s'
==
  if (a[n] == A)
  then if (s>0) then s' == s + a[n] else s' == a[n] && m' == n
  else s' == s + a[n]
```

D'où le programme, vérifié par le moteur d'inférence de Dafny :

```
method maxsumab(a: array<int>, A: int, B: int) returns (k: int, l: int)
  requires a.Length > 0
  ensures 0 <= k <= l < a.Length
  ensures forall i,j :: 0 <= i <= j < a.Length &&
                        a[i] == A && a[j] == B
                        ==> sum(a,i,j) <= sum(a,k,l)
{
  var n, r, m, s := 1, a[0], 0, a[0];
  k, l := 0, 0;
  while (n < a.Length)
    invariant 0 <= k <= l < n <= a.Length
    invariant r == sum(a, k, l)
    invariant forall i,j :: 0 <= i <= j < n && a[i] == A && a[j] == B
                            ==> sum(a, i, j) <= r
    invariant 0 <= m < n
    invariant s == sum(a, m, n-1)
    invariant forall i :: 0 <= i < n && a[i] == A ==> sum(a, i, n-1) <= s
    decreases a.Length - n
  {
    if (a[n] == A) {
        if (s > 0) {
            s := s + a[n];
        } else {
            s := a[n];
            m := n;
        }
    } else {
        s := s + a[n];
    }

    if (a[n] == B && r < s) {
        r := s;
        k := m;
        l := n;
    }

    n := n+1;
  }
}
```

10 Recherche binaire

```
 f(i:0 ≤ i < N) : array of int {N ≥ 1}
 {(∀i, j : 0 ≤ i < j < N : f.i ≤ f.j)}
;"recherche binaire"
 {present = (∃i : 0 ≤ i < N : f.i = X)}
```

Pour trouver si X est présent dans f, on cherche un entier i tel que $f.i = X$. Mais cette relation est trop contraignante car X peut ne pas être dans f. Dans ce cas, i peut indiquer où insérer X pour maintenir f trié : $f.i < X < f.(i+1)$. X pouvant être ou ne pas être dans f, il faut unir ces deux relations : $f.i \leq X < f.(i+1)$. Cette relation est encore trop contraignante car un X trop grand ou trop petit ne sera pas encadré par des éléments de f. Dans le cas d'un X trop grand, en adoptant la convention $f.N = \infty$, la relation est établie pour $i = N - 1$. Pour le cas d'un X trop petit, la relation est affaiblie avec la disjonction d'un prédicat supplémentaire :

R : $\quad f.i \leq X < f.(i+1) \quad \vee \quad Q$

Q : $\quad (\forall i : 0 \leq i < N : f.i > X)$

Puisque le tableau f est trié, une fois R établie, la postcondition du programme de recherche binaire est atteinte avec :

```
 present := (f.i=X)
```

On introduit un invariant en affaiblissant R par le remplacement de $i+1$ par une variable j. La solution sera atteinte lorsque $j = (i+1)$.

P : $\quad 0 \leq i < j \leq N \quad \wedge \quad (f.i \leq X < f.j \quad \vee \quad Q)$

```
 i,j := 0, N {P}
;do j ≠ (i+1) ->
    "réduire j-i en maintenant P"
 od {R}
;present := (f.i=X)
```

Il existe un entier h tel que $i < h < j$. La différence $j - i$ est réduite aussi bien par `i:=h` que `j:=h`. Il faut choisir l'affectation qui maintient P.

$$
\begin{array}{l}
\quad P[i\backslash h] \\
= \\
\quad 0 \leq h < j \leq N \quad \wedge \quad (f.h \leq X < f.j \quad \vee \quad Q) \\
\Leftarrow \{P, \quad i < h < j\} \\
\quad f.h \leq X
\end{array}
$$

De même pour $P[j\backslash h]$. D'où :

```
 if f.h ≤ X -> i := h
   |X < f.h -> j := h
 fi
```

Il faut choisir h tel que $i < h < j$. Par exemple, `h:=i+1` ou `h:=j-1` sont corrects. Dans le pire des cas, ces choix conduisent au résultat en N itérations. Le choix `h:= (i+j) div 2` (avec `div` dénotant la division entière) garantit un nombre d'itérations inférieur à $log(N)$. En fait, il est préférable de choisir `h:= i + (j-i) div 2` pour ne pas risquer de provoquer un débordement d'entier.

```
 f(i:0 ≤ i < N) : array of int {N ≥ 1}
 {(∀i, j : 0 ≤ i < j < N : f.i ≤ f.j)}
;present: bool
;i,j,h: int
;i,j:= 0, N
;do j ≠ (i+1) ->
    h:= i + (j-i) div 2
  ;if f.h ≤ X -> i := h
     |X < f.h -> j := h
    fi
 od
;present:= (f.i=X)
```

Dans ce programme, `f` n'apparaît que sous les formes `f.h` et `f.i`. Or :

$0 \leq i < h < j \leq N \quad \Rightarrow \quad 0 < h < N \ \wedge \ 0 \leq i < N$

Donc, `f.N` n'apparaît jamais au cours du calcul et la convention `f.N` $= \infty$ est bien fondée.

11 Recherche binaire en Dafny

```
method bsearch(a: array<int>, X:int) returns(r: int)
  requires a.Length > 0
  requires forall i,j :: 0<=i<j<a.Length ==> a[i]<=a[j]
  ensures 0<=r<a.Length-1 ==> a[r]<=X<a[r+1]
  ensures r==a.Length-1 ==> a[r]<=X
  ensures r==-1 ==> X<a[0]
{
    var i: nat, j: nat := 0, a.Length;
    var h: nat;
    if (X < a[0]) { return -1; }
    while (j != i+1)
      invariant 0 <= i < j <= a.Length
      invariant j < a.Length ==> a[i] <= X < a[j]
      invariant j == a.Length ==> a[i] <= X
      decreases j-i
    {
      h := i + (j-i)/2;
      if (a[h] <= X) {
        i:= h;
      } else { // X < a[h]
        j:= h;
      }
    }
    return i;
}
```

12 Plus longue sous-séquence croissante

`f(i:0 ≤ i < N) : array of int` $\{N \geq 1\}$

Une sous-séquence de taille s, avec $0 \leq s \leq N$, est obtenue en retirant $N - s$ éléments de f tout en conservant pour les s éléments restants leur ordre relatif dans f. Il y a 2^N sous-séquences. Comment atteindre la post-condition ci-dessous.

R : $k =$ longueur max d'une sous-séquence croissante (ssc) de f

Puisque chaque élément de f doit être lu au moins une fois, on propose un invariant en remplaçant la constante N par une variable n.

$P0$: $1 \leq n \leq N$

$P1$: $k =$ longueur max d'une ssc de $f(i : 0 \leq i < n)$

Pour maintenir P après la lecture d'une nouvelle valeur $f.n$, il faut décider si k doit ou non être incrémenté.

```
  {P1  ∧  1 ≤ n < N}
  if m ≤ f.n -> k:= k+1
    |m > f.n -> skip
  fi
  {P1[n\n+1]}
 ;n:= n+1
```

La nouvelle variable m doit être contrôlée par un invariant.

$P2$: $m =$ le plus petit des éléments en dernière position d'une ssc de longueur k dans $f(i : 0 \leq i < n)$

```
  n, k, m:= 1, 1, f.0 {P0 ∧ P1 ∧ P2}
 ;do n ≠ N ->
    {P1 ∧ P2 ∧ 1 ≤ n < N}
    if m ≤ f.n -> k:= k+1 ; m:= f.n
      |m > f.n -> ...
    fi
   ;n:= n+1
  od
```

Dans la seconde branche de l'alternative, la connaissance de $f.n$ permet-elle de diminuer m ?

```
{f.n < m}
if m' ≤ f.n -> m:= f.n
  |f.n < m' -> skip
fi
```

La nouvelle variable m' doit être contrôlée par un invariant.

$P2'$: $m' =$ le plus petit des éléments en dernière position d'une ssc de longueur k-1 dans $f(i : 0 \leq i < n)$

Pour maintenir $P2'$, une variable m'' est nécessaire, etc. Ainsi, il faut introduire un tableau de variables, `m(j: 1 ≤ j ≤ k)`, contrôlé par un invariant :

$P2$: $(\forall j : 1 \leq j \leq k : m.j =$ le plus petit des éléments en dernière position d'une ssc de longueur j dans $f(i : 0 \leq i < n))$

L'ancien m devient $m.k$, m' devient $m.(k-1)$, etc.

```
  n, k, m.1 := 1, 1, f.0 {P0 ∧ P1 ∧ P2}
;do n ≠ N ->
    if m.k ≤ f.n -> k:= k+1 ; m.k:= f.n
      |m.1 > f.n -> m.1:= f.n
      |m.1 ≤ f.n < m.k ->
         "trouver j tel que m.(j-1) ≤ f.n < m.j"
       ;m.j:= f.n
    fi
   ;n:= n+1
  od
```

Un dernier invariant contrôle une recherche binaire pour trouver j.

$P3$: $m.i \leq f.n < m.j \quad \wedge \quad 1 \leq i < j \leq k$

```
  {m.1 ≤ f.n < m.k}
  i, j:= 1, k {P3}
;do i ≠ (j-i) ->
    h:= i + (j-i) div 2 {i < h < j}
   ;if m.h ≤ f.n -> i:= h
      |f.n < m.h -> j:= h
    fi {P3}
  od {m.(j − 1) ≤ f.n < m.j}
```