CHAPITRE 1

LOGIQUE, INVARIANTS ET VÉRIFICATION D'ALGORITHMES

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> Edsger W. Dijkstra

> One does not need to give a formal proof of an obviously correct program; but one needs a thorough understanding of formal proof methods to know when correctness is obvious.
>
> John C. Reynold

Problème:

- Comment concevoir un programme afin d'avoir une *vision* aussi *claire* que possible des situations dynamiques qu'il engendre lors de son exécution ?
- L'objectif de la programmation est une *lisibilité* aussi claire que possible du programme, afin de pouvoir le communiquer entre «programmeurs» et surtout de pouvoir fournir une démonstration plus ou moins rigoureuse de sa validité (faire la preuve qu'il fournira les résultats désirés).

Comment atteindre au mieux ces objectifs? Avec des techniques adéquates de programmation.

1.1 SPÉCIFICATION

Définition 1

On appelle **spécification** d'un programme un quadruplet formé de

- une désignation et un typage des données,
- une désignation et un typage des résultats,
- une **pré condition** définissant des contraintes sur les données restreignant les valeurs admises par le typage.
- une **post condition** définissant le résultat en fonction des données.

Exemple 2

Voici un modèle pour le problème de calcul de la racine carrée entière par défaut.

- Donnée: `n` entier (nombre dont on veut calculer la racine).
- Résultat: `r` entier (racine carrée entière par défaut de `n`).
- Pré condition: $n \geq 0$.
- Post condition:
$$r^2 \leq n < (r+1)^2.$$

Définition 3

Un programme est **totalement correct** s'il calcule le résultat attendu et termine.

Définition 4

Un programme est **partiellement correct** s'il calcule le résultat attendu sous l'hypothèse que toutes ses itérations se terminent.

1.2 EXÉCUTION SYMBOLIQUE

§1 Triplet de Hoare

Définition 5

L'exécution d'un programme A sur une valeur symbolique de données définies par l'expression p qui donne une valeur symbolique résultat définie par l'expression q est notée

$$\{\ p\ \}\quad A\quad \{\ q\ \}.$$

Cette notation, appelée **triplet de Hoare**, signifie que si on exécute le programme A sur n'importe quelle données vérifiant l'assertion $\{\ p\ \}$, alors on obtient une valeur vérifiant l'assertion $\{\ q\ \}$ si le programme termine.

Exemple 6

```
{n == n0}    n = n - 1    {n == n0 - 1}
```

Test 7

Le triplet de Hoare suivant est-il valide?

```
{n > 0}    n = n + 1    {n >= 0}
```

Exercice 1. Déterminer les triplet de Hoare valides. On supposera que `i`, `j` et `N` sont des entiers.

1. { $i == 1$ } `j = i` { $i == j == 1$ }.
2. { $i == 1$ } `i = j` { $i == j == 1$ }.
3. { $0 \leq i < N$ } `i = i + 1` { $0 < i \leq N$ }.
4. { `True` } `i = j + 1` { $i < j$ }.
5. { $i == 1$ } `i = 0` { `True` }.
6. { $i == 0$ } `i = 1` { `False` }.
7. { `False` } `i = 1` { $i == 0$ }.

§2 Règle d'affectation

Théorème 8

Règle d'affectation simple
Étant donnée une assertion Q

$$\{ Q[V \mapsto E] \} \quad V = E \quad \{ Q \}$$

est un triplet de Hoare valide.

Pour l'instant, considérons que $Q[V \mapsto E]$ signifie «l'assertion Q dans laquelle nous avons substituer E à chaque occurrence de V».

Exercice 2. Déterminer la plus faible précondition P pour que les triplets suivants soient valides dans l'axiomatique de Hoare. Toutes les variable sont supposées de type `int`.

1. `{P} x = 0 {x == 0}`
2. `{P} x = 0 {x >= 0}`
3. `{P} x = 0 {x > 0}`
4. `{P} x = 0 {y > 0}`
5. `{P} x = x + 1 {x == 1}`
6. `{P} x = x + 1 {x > 0}`
7. `{P} x = x + 1 {x == y}`
8. `{P} x = x - 1 {x == y - 1}`

§3 Implications

Théorème 9

On suppose

$$\{ P \} \quad \mathtt{A} \quad \{ Q \}.$$

Si de plus, $P' \implies P$ *et* $Q \implies Q'$, *alors*

$$\{ P' \} \quad \mathtt{A} \quad \{ Q' \}.$$

On dit que l'on peut **renforcer** la pré condition ou **relacher** la post condition.

Exemple 10

On a

```
{n - 1 >= 0}    n = n - 1    {n >= 0}
```

Si n est un entier, alors on a l'implication

$$n > 0 \implies n - 1 \geq 0$$

et donc

```
{n > 0}    n = n - 1    {n >= 0}
```

Exercice 3. Les affirmation de correction suivantes sont-elles valides?

1. `{n < 0} n = n * n {n > 0}`
2. `{n != 0} n = n * n {n > 0}`
3. `{True} n = n * n {n >= 0}`
4. `{False} n = n * n {n < 0}`

§4 Composition séquentielle

Théorème 11

Si $\{ P \} C_0 \{ Q \}$ *et* $\{ Q \} C_1 \{ R \}$ *alors*

$$\{ P \} \quad C_0; C_1 \quad \{ R \}$$

Exemple 12

Montrer la validité du code suivant

```
1 # x == x0 et y == y0
2 z = x
3 x = y
4 y = z
5 # x == y0 et y == x0
```

Exercice 4. Vérifier la validité de

```
# i ≥ 0 et y == x^i
y = y * x
i += 1
# i ≥ 0 et y == x^i
```

Exercice 5. Vérifier la validité de

```
# k > 0 et f == Fib(k) et g == Fib(k − 1)
t = f + g
g = f
f = t
# k >= 0 et f == Fib(k + 1) et g = Fib(k)
```

où $Fib(n)$ désigne le terme d'indice n de la suite de Fibonacci, définie par $Fib(0) = 0$, $Fib(1) = 1$ et pour tout $n > 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$.

Exercice 6. Quelle pré condition P permet au code suivant d'être correct?

```
# P
t = f + g
g = f
f = t
k = k + 1
# k > 0 et f == Fib(k) et g == Fib(k − 1)
```

Exercice 7. Les implémentation alternative des spécification de l'exercice **5** sont-elles valides?

1.

```
# k > 0 et f == Fib(k) et g == Fib(k − 1)
t = f
f = f + g
g = t
# k >= 0 et f == Fib(k + 1) et g = Fib(k)
```

2.

```
# k > 0 et f == Fib(k) et g == Fib(k − 1)
f = f + g
g = f - g
# k >= 0 et f == Fib(k + 1) et g = Fib(k)
```

Exercice 8. Les assertions de corrections suivantes sont elles correctes? Si oui, en donner une preuve; si non, donner la plus faible pré condition nécessaire pour avoir un code correct.

```
# x == x0 et y == y0
x = x - y
y = x + y
x = y - x
# x == y0 et y == x0
```

§5 Conditionnelle

Théorème 13

On suppose

$$\{ P \text{ et } B \}\, C \,\{ Q \} \qquad \text{et} \qquad P \text{ et } (\text{non } B) \implies Q.$$

Alors

$$\{ P \} \quad \boxed{\texttt{if B:}\ \texttt{C}} \quad \{ Q \}$$

Ou plus généralement,

Théorème 14

On suppose

$$\{ P \text{ et } B \}\, C_0 \,\{ Q \} \qquad \text{et} \qquad \{ P \text{ et } (\text{non } B) \}\, C_1 \,\{ Q \}.$$

Alors

$$\{ P \} \quad \boxed{\begin{array}{l}\texttt{if B:}\\ \quad \texttt{C\_0}\\ \texttt{else:}\\ \quad \texttt{C\_1}\end{array}} \quad \{ Q \}$$

On a des résultat analogue avec l'utilisation de `elif`, qui n'est qu'une concaténation d'un bloc `else` avec un bloc `if`.

On peut présenter le résultat sous cette forme.

```
# {P}
if B:
    # {P et B}
    C_0
    # {Q}
else:
    # {P et non B}
    C_1
    # {Q}
# {Q}
```

Exemple 15

Nous souhaitons écrire une fonction qui affecte à la variable `m` le plus grand entier parmi `x` et `y`.

- `x`, `y`, `m` sont des entiers.
- Pré condition: aucune
- Post condition: $m = \max(x, y)$, c'est-à-dire

$$m \geq x \text{ et } m \geq y \text{ et } (m = x \text{ ou } m = y).$$

On propose le code suivant

```
1 if x > y:
2     m = x
3 else:
4     m = y
```

Nous devons gérer les deux cas, c'est-à-dire démontrer

1.
```
# {True and x > y}
    m = x
# {m >= x and m >= y and (m == x or m == y)}
```

2.
```
# {True and (not x > y)}
    m = y
# {m >= x and m >= y and (m == x or m == y)}
```

1.3 Preuve des segments itératifs

§1 Invariant de boucle

À première vue, il peut sembler difficile de traiter le cas des boucle. On pourrait croire qu'il faudrait considérer chaque itération séparement!

En pratique, la clef pour raisonner sur un programme avec une boucle while est d'exhiber une assertion bien choisie, appelée **invariant de boucle**. Intuitivement, un invariant de boucle décrit «l'état intermédiaire» d'une boucle dont on a déjà parlé. On doit toujours avoir conscience d'un invariant en écrivant une boucle, même s'il n'est pas totalement formalisé.

Une fois bien compris le concept d'invariant et comment les obtenir, les invariant ne servent pas seulement à *vérifier* le code, mais permettent d'*écrire* un code correct par construction!

Théorème 16

On suppose

$$\{ I \text{ et } B \} \quad C \quad \{ I \}$$

Alors

$\{ I \}$ `while B: C` $\{ I \text{ et non } B \}$

On peut présenter le résultat sous cette forme:

```
# {I}
while B:
    # {I and B}
    C
    # {I}
# {I and not B}
```

Remarquons que l'invariant apparait quatre fois:

- L'assertion I doit être *préservée* par chaque exécution du code `C`, le corps de la boucle, lorsque `B`, la condition de boucle, est vraie. Ainsi, I est à la fois une pré condition et une post condition pour le corps de la boucle!

  Remarquons toutefois que I peut être faux *temporairement* durant l'exécution de `C`.

- L'assertion I est la pré condition de la boucle, et doit donc être vraie *avant* l'exécution de la boucle. En pratique, l'invariant découle souvent des spécifications, avec l'ajout éventuel d'un pré traitement des données.

- L'assertion I et la négation de B sont alors vraies *après* l'exécution de la boucle, indépendamment du nombre d'exécution du corps de la boucle (éventuellement aucune).

Dans de nombreux exemples, l'assertion «I et non B» implique la post condition spécifiée. Parfois, il est nécessaire d'ajouter un post traitement pour finaliser le programme.

Exemple 17

- `x` et `y` sont des flottants. `i` et `n` sont des entiers.
- Pré condition: aucune.
- Post condition: $y = x^n$.

On complétera le code suivant:

```
#

while i != n:
    y = y * x
    i = i + 1
# y == x^n
```

§2 Terminaison d'une boucle

Dans l'exemple précédent,

```
i = 0
y = 1
while i != n:
    # Invariant: i ≥ 0 et y == x^i
    y = y * x
    i = i + 1
# y == x^n
```

l'exécution pour $n < 0$ ne termine pas. Si nous modifions la pré condition en $n \geq 0$, alors il est facile de vérifier que le code précédent termine. La variable `i` est initialisée à 0, et sa valeur est incrémentée (de 1) à chaque itération. La condition de boucle `i != n`, avec $n \geq 0$, finira par devenir fausse et la boucle terminera. On en déduit que le code est correct pour tout $n \geq 0$.

Dans d'autres exemples, il est plus dur de vérifier la terminaison de l'algorithme. On utilise pour cela une **fonction de terminaison** ou **variant de boucle**.

Définition 18

Un **variant** de boucle est une expression *entière* N telle que, pour chaque exécution du corps de la boucle,

- $N > 0$ initialement,
- N décroit *strictement* durant l'exécution.

En pratique, on ajoute une condition correspondant à $N > 0$ dans l'invariant de boucle.

Exemple 19

Ajoutons la condition `i <= n` à l'invariant précédent (l'inégalité large n'est pas une erreur). Le nouvel invariant devient

$$0 \leq i \leq n \quad \text{et} \quad y = x^i.$$

Dans cet exemple, le variant de boucle est $n - i$. Il est facile de voir que lorsque ce nouvel invariant est vrai ainsi que la condition de boucle ($i \neq n$) alors $n - i > 0$. De plus, chaque exécution du corps de boucle incrément `i` de 1, et fait donc décroitre strictement $n - i$.

§3 Exercices

Exercice 9. Utiliser l'invariant proposé pour prouver que, après exécution de ce code, la variable `f` a pour valeur $Fib(n)$.

```
f, g, k = 1, 0, 1
while k != n:
    # Invar (k > 0 et f == Fib(k) et g == Fib(k - 1))
    t = f + g
    g = f
    f = t
    k = k + 1
```

Exercice 10. Utiliser l'invariant proposé pour prouver que cette fonction est (partiellement) correcte.

```python
def racine_int(n):
    """Retourne la racine carrée entière par défaut d'un entier.

    :n: nombre dont on veut calculer la racine (entier >= 0)
    :returns: r : racine carrée entière par défaut (entier >= 0)

    Pré condition : n >= 0
    Post condition : r ** 2 <= n et n < (r + 1) ** 2
    """
    r = 0
    while (r + 1) ** 2 <= n:
        # Invariant: r ** 2 <= n
        r = r + 1
    return r
```

Exercice 11. Effectuer la preuve totale de l'algorithme suivant, qui affecte la somme des éléments de $b_1, \ldots, b_{10}$ à `s`.

```python
i, s = 10, 0
while i != 0:
    s = s + b[i]
    i = i - 1
```

Precondition	`b` est une liste de nombre de longueur >10
Invariant	$P : 0 \leq i \leq 10$ et $s = \sum_{i+1 \leq k \leq 10} b[k]$
Variant	$t : i$
Postcondition	$R : s = \sum_{1 \leq k \leq 10} b[k]$

Exercice 12. Effectuer la preuve totale de l'algorithme suivant qui affecte à i la plus grande puissance de 2 inférieure à n.

```python
i = 1
while 2 * i <= n:
    i = 2 * i
```

Precondition	$0 < n$
Invariant	$P : 0 < i \leq n$ et $\exists p, i = 2^p$
Variant	$t : n - i$
Postcondition	$R : 0 < i \leq n < 2 * i$ et $\exists p, i = 2^p$

Exercice 13. Le n-ième nombre de Fibonacci f_n ($n > 0$) est défini par $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour $n > 1$. Effectuer la preuve totale de l'algorithme suivant qui calcule le n-ième nombre de Fibonacci.

```
i, a, b = 1, 1, 0
while i < n:
    i = i + 1
    a, b = a + b, a
```

Precondition	$n > 0$
Invariant	$P : 0 \leq i \leq n$ et $a = f_i$ et $b = f_{i-1}$
Variant	$t : n - i$
Postcondition	$R : a = f_n$

Exercice 14. Effectuer la preuve totale de l'algorithme suivant qui calcule le quotient q et le reste r dans la division euclidienne de x par y.

```
q, r = 0, x
while r >= y:
    r = r - y
    q = q + 1
```

Precondition	$x \geq 0$ et $0 < y$
Invariant	$P : 0 \leq r$ et $0 < y$ et $q * y + r = x$
Variant	$t : r$
Postcondition	$R :$ (compléter)

Exercice 15. Effectuer la preuve totale de l'algorithme suivant qui détermine la valeur maximale dans une liste `A`.

```
def maximum(A):
    n = len(A)
    m = A[0]
    for i in range(1, n):
        if A[i] > m:
            m = A[i]
    return m
```