Chapitre 3 : Vérification de programme

Table des matières

1 Terminaison

1.1 Définition (terminaison)

On dit qu'un programme / algorithme termine s'il n'éxécute qu'un nombre fini d'opérations quelle que soit l'entrée.

1.2 Remarque

Le nombre d'opérations peut être arbitrairement grand en fonction de l'entrée (pas de majorant commun à toutes les entrées).
Si $\exists c \mid \forall n,\ C(n) \leqslant c \longrightarrow O(1)$

1.3 Exemple

Montrons que les deux fonctions suivantes terminent :

```
let is_prime (n : int) : bool =
    let res = ref true in
    for i = 2 to n - 1 do
        if n mod i = 0 then
            res := false
    done;
    !res
```

La boucle est éxécutée $n-2$ fois, et contient uniquement des opérations élémentaires qui terminent.

```
let count_primes (n : int) : int =
    let res = ref 0 in
    for i = 2 to n do
        if is_prime i then incr res
    done;
    !res
```

La boucle est éxécutée $n-1$ fois, chaque appel à `is_prime` termine, et le reste est composé d'opérations élémentaires qui terminent.

Conclusion : une boucle inconditionnelle termine dès que les opérations qui le composent terminent.

Attention : une boucle `for` en C est une boucle `while` déguisée.

Rappel

```
for (c1 ; c2 ; c3) c4
//Is equivalent to
{
    c1;
    while (c2) {
        c4;
        c3;
    }
}
```

La difficulté dans les preuves de terminaison réside dans les boucles `while` et les fonctions récursives.

1.4 Définition (variant)

Un variant est une quantité entière positive au début de chaque itération / appel récursif et telle que sa valeur décroît strictement au cours de l'itération.

Généralisation : on peut autoriser les quantités négatives strictement croissantes.

1.5 Théorème

Une boucle composée d'opérations qui terminent et admettant un variant termine.

☐ Il n'existe pas de suite strictement décroissante d'entiers positifs. ■

$\nexists (u_n) \in \mathbb{N}^{\mathbb{N}} \mid \forall n \in \mathbb{N},\ u_{n+1} < u_n$

$\forall (u_n) \in \mathbb{N}^{\mathbb{N}},\ \exists n \in \mathbb{N} \mid u_{n+1} \geqslant u_n.$

1.6 Exemple

```
while (a > 0 && b > 0) {
    if (a % 2 == 0) {
        a++;
        b -= 3;
    }
    else {
        a -= 2;
        b++;
    }
}
```

- Si $a \leqslant 0$ ou $b \leqslant 0$, on n'entre pas dans la boucle.
- Si $a > 0$ et $b > 0$,
  - `a` est un variant à partir de la deuxième itération (`a` pair $\rightarrow$ `a + 1` impair puis des soustractions avec 2 conservent la parité)
  - `a + b` est un variant.

1.7 Cas des fonctions récursives

```
let rec is_even (n : int) : bool =
    match n with
    | 0 -> true
  (* | 1 -> false *)
    | _ -> is_even (n - 2)
```

Le variant "naturel" (n) peut devenir négatif si on oublie un cas d'arrêt.

Trouver une quantité strictement décroissante ne suffit pas, il faut vérifier que l'on atteint toujours un cas d'arrêt.

En cas de doute, on peut toujours écrire une fonction récursive comme une boucle `while`.

On parle de dérécursivation (H.P. dans sa généralité), simple dans le cas des fonctions récursives terminales.

```
let is_even (n : int) : bool =
    let m = ref n in
    while !m <> 0 && !m <> 1 do
        m := !m - 2
    done;
    !m = 0

```

2 Correction

2.1 Définition (correction)

On dit qu'un algorithme / programme est correct vis-à-vis d'une spécification si, quelle que soit l'entrée, il termine et renvoit un résultat qui respecte la spécification.

On parle de *correction totale* par contraste avec la *correction partielle*, définie comme suit :
Un programme / algorithme est *partiellement* correct vis-à-vis d'une spécification si, quelle que soit l'entrée *telle que le programme termine*, le résultat renvoyé respecte la spécification.

2.2 Exemple

```
let rec sum (n : int) : int =
    if n = 0 then 0
    else n + sum (n - 1)
```

Partiellement correct

```
let rec sum (n : int) : int =
    assert (n >= 0);
    if n = 0 then 0
    else n + sum (n - 1)
```

Spécification : $\forall n \in \mathbb{Z}$, `sum` vaut $\sum_{k=0}^{n} k$.

Termine bien dans tous les cas, mais ne renvoit pas de résultat dans le cas $n < 0$.

2.3 Définition (invariant)

Un invariant est un prédicat portant sur les variables du programme, qui est satisfait avant d'entrer dans la boucle / au début du premier appel (récursif) et tel que l'éxécution d'une itération conserve sa véracité.

2.4 Remarque

L'objectif est de trouver un invariant tel que sa véracité en sortie de boucle guarantit la spécification.

2.5 Exemple

```
let suplog2 (n : int) : int =
    let k = ref 0 in
    let p = ref 1 in
    while !p < n do
        incr k;
        p := 2 * !p
    done;
    !k

```

Terminaison : `n - !p` est un variant.

Correction : on montre que $I(k,p) = (2^{k-1} < n \text{ et } p = 2^k)$ est un invariant :

- Avant d'entrer dans la boucle, $I(0,1) = (2^{-1} < n \text{ et } 1 = 2^0)$: ok.
- Si $I(k,p)$ est vrai au début d'une itération, on veut montrer $I(k+1,2p)$:

  $I(k+1,2p) = (2^k < n \text{ et } 2p = 2^{k+1})$

  - $2^k < n$: vrai car $2^k = p < n$ puisque la condition de boucle était vraie au début de l'itération ;
  - $2p = 2^{k+1}$: vrai car $p = 2^k$ d'après $I(k,p)$.

I en sortie de boucle démontre la correction : on a $I(k,p) = (2^{k-1} < n \text{ et } p = 2^k)$ et la condition de boucle est fausse, *i.e* $p \geqslant n$ donc $2^k \geqslant n$.

Attention, cette démonstration est fausse !
Ex : $n = \underbrace{\langle 0111\ldots1\rangle_2}_{\text{nombre de bits du type } \texttt{int}}$ (type d'entiers signés)
p : $\langle 0\ldots01\rangle_2$ puis $\langle 0\ldots010\rangle_2$ puis … puis $\langle 01\ldots0\rangle_2 < n$
Or si $p = \langle 01\ldots0\rangle_2$, alors $2p < 0 < n$

Il peut y avoir des dépassement de capacité : il faut en tenir compte dans la spécification, ou bien détecter l'*overflow* dans le calcul.

2.6 Exercice : suite de Fibonacci

```
int fibo(int n) {
    int x = 1, y = 1;
    for (int i = 0 ; i < n - 1 ; i++) {
        y = x + y;
        x = y - x;
    }
    return y;
}
```

On note $(F_n)_{n\in\mathbb{N}}$ les termes de la suite de Fibonacci.
$I(i,x,y) = (y = F_{i+1} \text{ et } x = F_i)$

3 Graphe de flot de contrôle

3.1 Motivation

En pratique, il est difficile de trouver des invariants pour démontrer la correction d'un programme. Il est possible de démontrer des propriétés sur un programme à l'aide d'outils qui analysent le code sans l'exécuter (on parle d'analyse statique), mais ils ne permettent pas de tout démontrer : on fait en général des tests.

« *Program testing can be used to show the presence of bugs, but never to show their absence !* » Edsger W. Dÿkstra

« *Beware of bugs in the above code ; I have only proven it correct, not tried it.* » Donald E. Knuth (ex : compilateur bugé, ...)

Le choix des tests est particulièrement important pour ne pas "manquer" des erreurs.

Une analyse de la structure du code peut aider à les concevoir.

3.2 Graphe de flot de contrôle

Définition : Un graphe de flot de contrôle est un graphe orienté représentant la structure d'un programme. Il possède un point d'entrée et un point de sortie correspondant à l'entrée et à la sortie du programme. Ses sommets sont de deux natures : il y a des blocs d'instructions élémentaires et des noeuds de décision étiquetés par une condition et correspondant aux branchements dans l'exécution du programme. Les arcs du graphe relient les noeuds correspondant à des blocs d'instructions et branchements consécutifs.

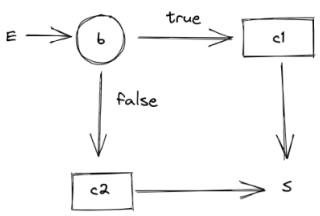
Construction :

- Les instructions élémentaires consécutives sont réunies dans un même bloc

Ex : E → S

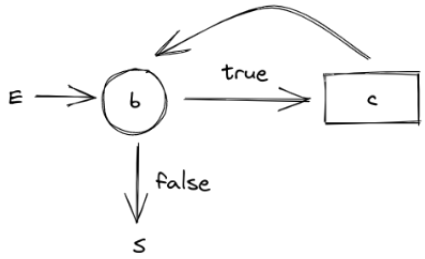
```
1 int x = 1;
2 int y = 2;
3 x = x + y;
4 y = x - y;
5 x = x - y;
```

- Une instruction conditionnelle crée un branchement : `if b then c1 else c2`



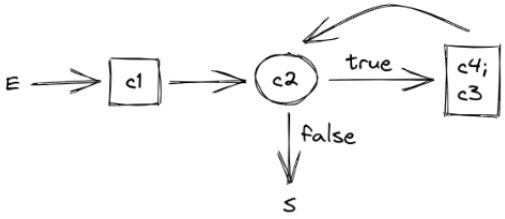
– Une boucle crée également un branchement :

```
while (b) {
    c
}
```



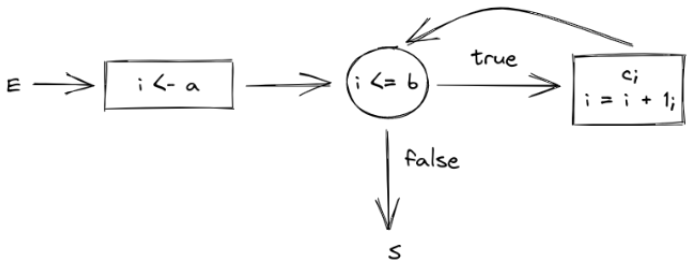
On pourra voir une boucle `for` comme une boucle `while`

```
for (c1; c2; c3)
    c4;
```



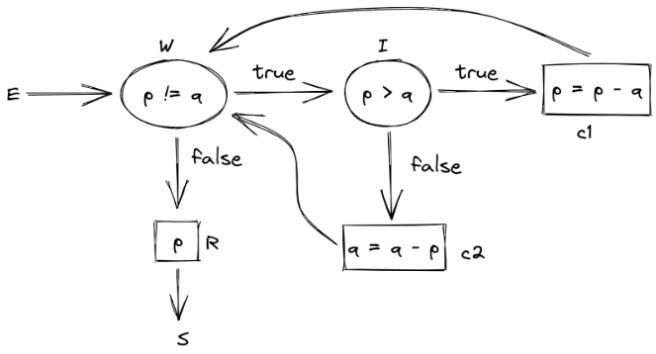
On peut adopter cela en OCaml en écrivant une condition qui porte sur la variable de boucle.

```
for i = a to b do
    c
done
```



Ex : calcul du PGCD par soustractions succesives

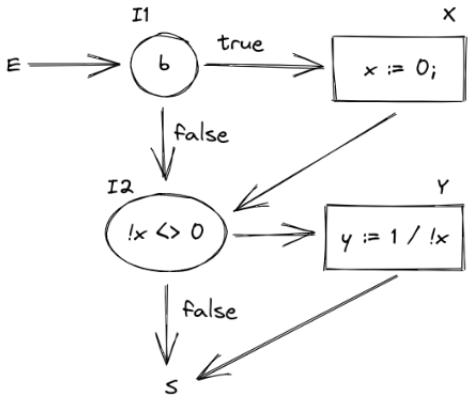
```
while (p != q) {
    if (p > q)
        p = p - q;

    else
        q = q - p;
}

return p;
```



3.3 Chemins et exécutions

Une éxécution du programme correspond à un chemin dans son graphe de flot de contrôle.
Ex : PGCD avec $p = 6$ et $q = 10$, on obtient le chemin `E` → `W` → `I` → `c2` → `W` → `I` → `c1` → `W` → `I` → `c2` → `W` → `R` → `S`
Un chemin ne correspond pas forcément à une exécution.
Par exemple :

```
1 if b then x := 0;
2 if !x <> 0 then y := 1 / !x
```



Le chemin `E` → `I1` → `X` → `I2` → `Y` → `S` ne correspond à aucune exécution.

On appelle chemin faisable un chemin qui correspond à (au moins) une exécution.
Le principe de la conception de tests à l'aide du graphe de flots de contrôle est le suivant :

1) On choisit un ensemble de chemins selon un critère de couverture fixée à l'avance ;

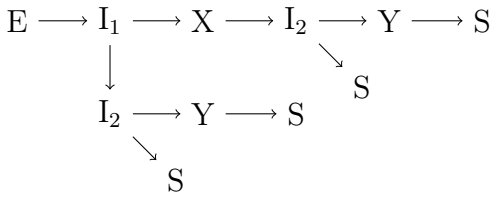
2) On élimine les chemins infaisables ;

3) Pour chaque chemin restant, on étudie les conditions qui portent sur les entrées du programme pour que l'exécution correspondante au chemin. On choisit alors une entrée comme cas de test.

3.4 Critère de couverture

Pour couvrir tous les cas possibles, le critère le plus évident est de prendre tous les chemins possibles.

Par exemple, pour le graphe en 3.3., on a les chemins suivants :



Cependant, dès qu'il y a une boucle, il y a une infinité de chemins dans le graphe.

Rq : on peut avoir une infinité de chemins faisables même si le programme termine (ex : une boucle dont le nombre d'itérations dépend d'une variable d'entrée)

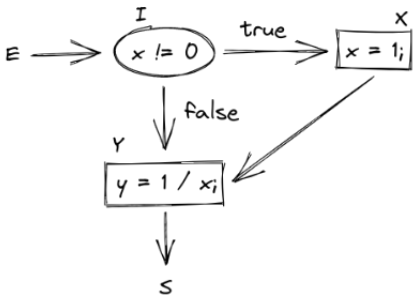
On peut affaiblir ce critère en considérant tous les chemins de longueur au plus k (fixé) ou tous les chemins qui vont parcourir la boucle au plus k fois.

On peut choisir d'autres critères, comme :

– La couverture de toutes les instructions du programme : on choisit un ensemble de chemins tels que chaque sommet est couvert par au moins un chemin.

Ex :

```
1 if (x != 0)
2     x = 1;
3 y = 1 / x;
```

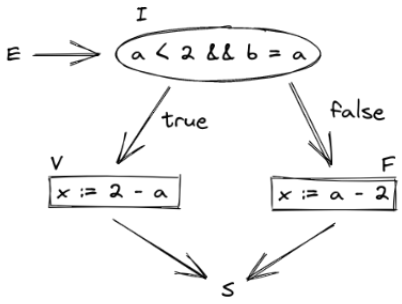


Le chemin `E` → `I` → `X` → `Y` → `S` satisfait le critère mais ne couvre pas la division par 0.

– La couverture de toutes les branches : on choisit un ensemble de chemins tels que pour chaque noeud de décision, les deux branches issues de ce noeud sont couvertes par au moins un chemin.

Ex :

```
1 if a < 2 && b = a then
2     x := 2 - a
3 else
4     x := a - 2
```



Le critère est satisfait par l'ensemble de chemins { `E` → `I` → `V` → `S`, `E` → `I` → `F` → `S` }.

Ces chemins sont faisables ($a = b = 1$ et $a = b = 3$)

La couverture des boucles implique la couverture des instructions mais pas nécessairement la couverture de toutes les possibilités (il manque le cas où $a \neq b$)

Il est possible de rajouter comme contrainte le choix des tests couvrant toutes les valeurs possibles par les sous-conditions, mais le nombre de tests devient exponnetiel en le nombre de sous-conditions.

Rq : il n'est pas forcément intéressant de considérer toutes les valeurs par les sous-conditions (ex : évaluation paresseuse des opérateurs booléens)

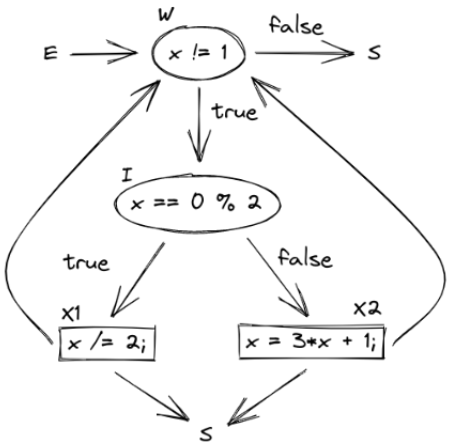
3.5 Détection des chemins infaisables et choix des tests

Afin de déterminer si un chemin est faisable et, le cas échéant, de choisir un test concret le réalisant, on peut calculer une "condition de chemin" en fonction de la valeur des entrées par exécution symbolique. Cette condition réunit les contraintes qui portent sur les entrées pour que l'exécution corresponde au chemin. On suit les étapes suivantes :

1) On donne une valeur symbolique aux variables d'entrée ;
2) On initialise la condition de chemin à `true`
3) On suit le chemin :
   - Si le noeud est un bloc d'instructions : on exécute ces instructions sur les valeurs symboliques.
   - Si le noeud est un noeud de décision : on remplace la condition de chemin par la conjonction de cette condition et de la condition du noeud (ou de sa négation suivant la branche suivie par le chemin) où l'on a remplacé les variables par leur valeur symbolique.

Exemple – suite de Syracuse :

```
1 while (x != 1) {
2     if (x % 2 == 0)
3         x /= 2;
4
5     else
6         x = 3*x + 1;
7 }
```



Chemin	E	$\rightarrow$	W	$\rightarrow$	I	$\rightarrow$	X1	$\rightarrow$	W	$\rightarrow$	S
x	x_0		–		–		$x_0/2$		–		
Condition de chemin	vrai		$x_0 \neq 1$		$\begin{cases} x_0 \neq 1 \\ x_0 \equiv 0\ [2] \end{cases}$		–		$\begin{cases} x_0 \neq 1 \\ x_0 \equiv 0\ [2] \\ x_0/2 = 1 \end{cases}$		–

Le chemin est faisable avec l'entrée $x = 2$.

Chemin	E	$\rightarrow$	W	$\rightarrow$	I	$\rightarrow$	X2	$\rightarrow$	W	$\rightarrow$	S
x	x_0		–		–		$3x_0 + 1$		–		
Condition de chemin	vrai		$x_0 \neq 1$		$\begin{cases} x_0 \neq 1 \\ x_0 \equiv 1\ [2] \end{cases}$		–		$\begin{cases} x_0 \neq 1 \\ x_0 \equiv 1\ [2] \\ 3x_0 + 1 = 1 \end{cases}$		–

Le chemin est infaisable : $\begin{cases} x_0 \equiv 1\ [2] \\ 3x_0 + 1 = 1 \end{cases}$: contradiction (x_0 impair donc $3x_0 + 1$ pair).