

# Preuve de programme

Quentin Fortier

June 13, 2022

# Preuve de programme

Soit  $f$  une fonction.

On veut montrer 2 choses :

- $f$  **termine** pour chaque entrée : ne fait pas boucle infinie ou appels récursifs infinis

# Preuve de programme

Soit  $f$  une fonction.

On veut montrer 2 choses :

- $f$  **termine** pour chaque entrée : ne fait pas boucle infinie ou appels récursifs infinis
- $f$  est **correct** : la valeur renvoyée par  $f$  est bien celle qu'on veut

Pour montrer qu'une boucle while (ou fonction récursive) termine :

- utiliser une suite d'entiers strictement décroissante (à chaque itération du while ou appel récursif)
- montrer que la boucle s'arrête lorsque la suite devient négative

---

```
let rec pgcd a b =  
  if b = 0 then a  
  else pgcd b (a mod b)
```

---

## Question

Comment montrer que `pgcd a b` termine si  $a \geq b$  ?

---

```
let rec pgcd a b =  
  if b = 0 then a  
  else pgcd b (a mod b)
```

---

## Question

Comment montrer que `pgcd a b` termine si  $a \geq b$  ?

Soient  $a_n$  et  $b_n$  les valeurs de  $a$  et  $b$  après  $n$  appels récursifs.

---

```
let rec pgcd a b =  
  if b = 0 then a  
  else pgcd b (a mod b)
```

---

## Question

Comment montrer que `pgcd a b` termine si  $a \geq b$  ?

Soient  $a_n$  et  $b_n$  les valeurs de  $a$  et  $b$  après  $n$  appels récursifs.

On montre par récurrence sur  $n$  que  $a_n \geq b_n$ ,  $a_n \searrow \searrow$  et  $b_n \searrow \searrow$ .

---

```
let rec pgcd a b =  
  if b = 0 then a  
  else pgcd b (a mod b)
```

---

## Question

Comment montrer que `pgcd a b` termine si  $a \geq b$  ?

Soient  $a_n$  et  $b_n$  les valeurs de  $a$  et  $b$  après  $n$  appels récurrents.

On montre par récurrence sur  $n$  que  $a_n \geq b_n$ ,  $a_n \searrow \searrow$  et  $b_n \searrow \searrow$ .

(Complexité :  $O(\log(a))$ )



---

```
let dicho t e =  
  let i = ref 0 and j = ref (Array.length t) in  
  let res = ref false in  
  while not !res && !i < !j do  
    let m = (!i + !j)/2 in  
    if t.(m) = e then res := true  
    else if t.(m) > e then j := m  
    else i := m  
  done;  
  !res
```

---

## Question

Donner un exemple où cette version (erronée) de la recherche par dichotomie ne termine pas.

```
let dicho t e =  
  let i = ref 0 and j = ref (Array.length t) in  
  let res = ref false in  
  while not !res && !i < !j do  
    let m = (!i + !j)/2 in  
    if t.(m) = e then res := true  
    else if t.(m) > e then j := m  
    else i := m  
  done;  
  !res
```

## Question

Donner un exemple où cette version (erronée) de la recherche par dichotomie ne termine pas.

dicho [|0|] 1 fait boucle infinie

---

```
let dicho t e =  
  let i = ref 0 and j = ref (Array.length t) in  
  let res = ref false in  
  while not !res && !i < !j do  
    let m = (!i + !j)/2 in  
    if t.(m) = e then res := true  
    else if t.(m) > e then j := m  
    else i := m + 1  
  done;  
  !res
```

---

## Question

Comment montrer que la boucle **while** termine ?

---

```
let dicho t e =  
  let i = ref 0 and j = ref (Array.length t) in  
  let res = ref false in  
  while not !res && !i < !j do  
    let m = (!i + !j)/2 in  
    if t.(m) = e then res := true  
    else if t.(m) > e then j := m  
    else i := m + 1  
  done;  
  !res
```

---

## Question

Comment montrer que la boucle **while** termine ?

Montrer que  $!j - !i$  décroît strictement

# Terminaison

$m$  est égal à  $\lfloor \frac{i+j}{2} \rfloor$  donc vérifie par définition :

$$\frac{i+j}{2} - 1 < m \leq \frac{i+j}{2}$$

# Terminaison

$m$  est égal à  $\lfloor \frac{i+j}{2} \rfloor$  donc vérifie par définition :

$$\frac{i+j}{2} - 1 < m \leq \frac{i+j}{2}$$

- Si  $e < t.(m)$  :

# Terminaison

$m$  est égal à  $\lfloor \frac{i+j}{2} \rfloor$  donc vérifie par définition :

$$\frac{i+j}{2} - 1 < m \leq \frac{i+j}{2}$$

- Si  $e < t.(m)$  : on remplace  $j$  par  $m$

$$m - i \leq \frac{i+j}{2} - i = \frac{j-i}{2} < j - i$$

- Si  $e > t.(m)$  :

# Terminaison

$m$  est égal à  $\lfloor \frac{i+j}{2} \rfloor$  donc vérifie par définition :

$$\frac{i+j}{2} - 1 < m \leq \frac{i+j}{2}$$

- Si  $e < t.(m)$  : on remplace  $j$  par  $m$

$$m - i \leq \frac{i+j}{2} - i = \frac{j-i}{2} < j - i$$

- Si  $e > t.(m)$  : on remplace  $i$  par  $m + 1$

$$j - (m + 1) < j - \frac{i+j}{2} = \frac{j-i}{2} < j - i$$



# Terminaison

$m$  est égal à  $\lfloor \frac{i+j}{2} \rfloor$  donc vérifie par définition :

$$\frac{i+j}{2} - 1 < m \leq \frac{i+j}{2}$$

- Si  $e < t.(m)$  : on remplace  $j$  par  $m$

$$m - i \leq \frac{i+j}{2} - i = \frac{j-i}{2} < j - i$$

- Si  $e > t.(m)$  : on remplace  $i$  par  $m + 1$

$$j - (m + 1) < j - \frac{i+j}{2} = \frac{j-i}{2} < j - i$$

Ainsi  $j - i$  est une suite d'entiers strictement décroissante donc qui devient négatif, ce qui termine la boucle **while**.

---

```
while !m <> !n do
    if !m > !n then m := !m - !n;
    else n := !n - !m
done
```

---

## Question

Est-ce que cette boucle **while** termine pour  $n$  et  $m$  dans  $\mathbb{N}^*$  ?

# Fonction d'Ackermann

---

```
let rec ack n p = match n, p with  
  | 0, p -> p + 1  
  | n, 0 -> ack (n - 1) 1  
  | n, p -> ack (n - 1) (ack n (p - 1))
```

---

## Question

Est-ce que cette fonction termine ?

## Définition

Un **ordre** sur un ensemble  $E$  est une relation binaire  $\preceq$  vérifiant :

- $\forall x \in E, x \preceq x$  (**réflexif**)

## Définition

Un **ordre** sur un ensemble  $E$  est une relation binaire  $\preceq$  vérifiant :

- $\forall x \in E, x \preceq x$  (**réflexif**)
- $\forall x, y \in E, x \preceq y$  et  $y \preceq x \implies x = y$  (**antisymétrique**)

## Définition

Un **ordre** sur un ensemble  $E$  est une relation binaire  $\preceq$  vérifiant :

- $\forall x \in E, x \preceq x$  (**réflexif**)
- $\forall x, y \in E, x \preceq y \text{ et } y \preceq x \implies x = y$  (**antisymétrique**)
- $\forall x, y, z \in E, x \preceq y \text{ et } y \preceq z \implies x \preceq z$  (**transitif**)

## Définition

Un **ordre** sur un ensemble  $E$  est une relation binaire  $\preceq$  vérifiant :

- $\forall x \in E, x \preceq x$  (**réflexif**)
- $\forall x, y \in E, x \preceq y$  et  $y \preceq x \implies x = y$  (**antisymétrique**)
- $\forall x, y, z \in E, x \preceq y$  et  $y \preceq z \implies x \preceq z$  (**transitif**)

Exemple :  $\leq_{\mathbb{N}}$  (comparaison des entiers de  $\mathbb{N}$ )

## Définition

Un **ordre** sur un ensemble  $E$  est une relation binaire  $\preceq$  vérifiant :

- $\forall x \in E, x \preceq x$  (**réflexif**)
- $\forall x, y \in E, x \preceq y \text{ et } y \preceq x \implies x = y$  (**antisymétrique**)
- $\forall x, y, z \in E, x \preceq y \text{ et } y \preceq z \implies x \preceq z$  (**transitif**)

Exemple :  $\leq_{\mathbb{N}}$  (comparaison des entiers de  $\mathbb{N}$ )

## Définition

Un ordre est **bien fondé** s'il n'existe pas de suite infinie strictement décroissante pour cet ordre.



## Ordre lexicographique

L'ordre lexicographique  $\preceq$  sur  $\mathbb{N}^* \times \mathbb{N}^*$  par :

$$(a_1, a_2) \preceq (b_1, b_2) \iff a_1 < b_1 \text{ ou } (a_1 = b_1 \text{ et } a_2 \leq b_2)$$

Exemples :  $(1, 4) \preceq (2, 3)$ ,  $(1, 4) \preceq (1, 6)$

## Ordre lexicographique

L'ordre lexicographique  $\preceq$  sur  $\mathbb{N}^* \times \mathbb{N}^*$  par :

$$(a_1, a_2) \preceq (b_1, b_2) \iff a_1 < b_1 \text{ ou } (a_1 = b_1 \text{ et } a_2 \leq b_2)$$

Exemples :  $(1, 4) \preceq (2, 3)$ ,  $(1, 4) \preceq (1, 6)$

## Théorème

L'ordre lexicographique est bien fondé.

Preuve : exercice laissé au lecteur

# Fonction d'Ackermann

---

```
let rec ack n p = match n, p with
  | 0, p -> p + 1
  | n, 0 -> ack (n - 1) 1
  | n, p -> ack (n - 1) (ack n (p - 1))
```

---

## Question

Montrer que  $\text{ack } n \ p$  termine pour tout entiers positifs  $n, p$ .

# Fonction d'Ackermann

---

```
let rec ack n p = match n, p with  
  | 0, p -> p + 1  
  | n, 0 -> ack (n - 1) 1  
  | n, p -> ack (n - 1) (ack n (p - 1))
```

---

## Question

Montrer que  $\text{ack } n \ p$  termine pour tout entiers positifs  $n, p$ .

Supposons qu'il y ait une infinité d'appels récurifs et appelons  $(n_k, p_k)$  les arguments du  $k$ ème appel récurif.

# Fonction d'Ackermann

---

```
let rec ack n p = match n, p with  
  | 0, p -> p + 1  
  | n, 0 -> ack (n - 1) 1  
  | n, p -> ack (n - 1) (ack n (p - 1))
```

---

## Question

Montrer que  $\text{ack } n \ p$  termine pour tout entiers positifs  $n, p$ .

Supposons qu'il y ait une infinité d'appels récurrents et appelons  $(n_k, p_k)$  les arguments du  $k^{\text{ème}}$  appel récurrent.

Clairement,  $n_k$  et  $p_k$  sont des entiers positifs.

De plus  $(n_k, p_k)$  décroît strictement pour  $\preceq$  : contradiction.

Pour prouver qu'une fonction est correcte (renvoie bien le bon résultat), on utilise presque toujours un **raisonnement par récurrence** :

- Boucle **while** : récurrence sur le nombre d'itération, ce qu'on appelle aussi **invariant de boucle**
- Fonction récursive : récurrence sur le nombre d'appels récursifs

# Correction

---

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

---

---

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

---

Récurrance forte sur  $\mathcal{H}(n)$  : `exp_rapide a n` renvoie  $a^n$



# Tri fusion

---

```
let rec tri = function
  | [] -> []
  | [e] -> [e]
  | l -> let l1, l2 = split l in
          fusion (tri l1) (tri l2);;
```

---

# Tri fusion

---

```
let rec tri = function
  | [] -> []
  | [e] -> [e]
  | l -> let l1, l2 = split l in
          fusion (tri l1) (tri l2);;
```

---

Récurrance forte sur

$\mathcal{H}(n)$  : tri l renvoie une liste triée ayant les mêmes éléments que l

---

```
let dicho t e =  
  let i = ref 0 and j = ref (Array.length t) in  
  let res = ref false in  
  while not !res && !i < !j do  
    (* Invariant de boucle : si e appartient à t, *)  
    (* alors e est entre les indices !i et !j *)  
    let m = (!i + !j)/2 in  
    if t.(m) = e then res := true  
    else if t.(m) > e then j := m  
    else i := m + 1  
  done;  
  !res
```

---

---

```
let russe a b =  
  let res = ref 0 in  
  let c = ref a and d = ref b in  
  while !d <> 0 do  
    if !d mod 2 = 0  
    then (c := !c * 2;  
          d := !d / 2)  
    else (res := !res + !c;  
          d := !d - 1)  
  done;  
  !res
```

---

## Exercice

Dire ce que fait cette fonction et le prouver en donnant un invariant de boucle.

# Induction structurelle

Les types récurifs en OCaml donnent naturellement un schéma de récurrence, appelé **preuve par induction structurelle**.

Les types récurifs en OCaml donnent naturellement un schéma de récurrence, appelé **preuve par induction structurelle**.

On peut ainsi prouver qu'une proposition/un programme  $\mathcal{P}$  est correct sur les listes en montrant:

$$① \mathcal{P}([])$$

$$② \mathcal{P}(1) \implies \forall e, \mathcal{P}(e::1)$$

On verra plus tard les arbres binaires, définis par :

---

```
type arbre = Vide | Noeud of arbre * arbre
```

---

On peut démontrer une proposition  $\mathcal{P}$  sur les arbres en montrant:

- 1  $\mathcal{P}(\text{Vide})$
- 2  $\mathcal{P}(g) \wedge \mathcal{P}(d) \implies \mathcal{P}(\text{Noeud}(r, g, d))$

# Triplet de Hoare (HP)

## Définition

Si  $P$  est une **précondition** (booléen),  $Q$  est une **postcondition** et  $c$  un programme, le **triplet de Hoare**  $\{P\} c \{Q\}$  signifie :

« Si la commande  $c$ , démarrée dans un état initial satisfaisant  $P$ , termine, alors l'état final satisfait  $Q$  »

Exemple :  $\{x = 3\} x := x + 1 \{x = 4\}$ .



# Triplet de Hoare (HP)

Règles de déduction de la logique de Hoare :

$$\{P\} \text{ skip } \{P\} \qquad \{Q[x \leftarrow a]\} x := a \{Q\}$$

$$\frac{\{P\} c_1 \{Q\} \quad \{Q\} c_2 \{R\}}{\{P\} c_1; c_2 \{R\}}$$

$$\frac{\{P \wedge b\} c_1 \{Q\} \quad \{P \wedge \neg b\} c_2 \{Q\}}{\{P\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{Q\}}$$

$$\frac{\{P \wedge b\} c \{P\}}{\{P\} \text{ while } b \text{ do } c \{P \wedge \neg b\}}$$

## Triplet de Hoare (HP)

Why3 est un logiciel de verification de programmes, basé sur la logique de Hoare et avec un langage proche d'OCaml :

# Triplet de Hoare (HP)

Why3 est un logiciel de verification de programmes, basé sur la logique de Hoare et avec un langage proche d'OCaml :

---

```
module SumGauss
  use int.Int
  use ref.Ref

  let sum_gauss (n: int) : int
    requires { true }
    returns  { r -> 2 * r = n * (n-1) }
  = let sum = ref 0 in
    for i = 1 to n - 1 do
      invariant { 2 * !sum = i * (i - 1) }
      sum := !sum + i
    done;
    !sum
end
```

---