Preuves de programme

Quentin Fortier

December 12, 2021

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

Terminaison

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

Terminaison

```
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 >= b` ?

Terminaison

```
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 >= b` ?

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

Terminaison

```
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 >= 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 $b_n \searrow\searrow$.

Terminaison

```
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 >= 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 $b_n \searrow\searrow$.

(Complexité : O$(\log(a))$)

Terminaison

```
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.

Terminaison

```
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

Terminaison

```
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 ?

Terminaison

```
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 $\lfloor \dfrac{i+j}{2} \rfloor$ 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`.

Terminaison

```
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 ?

Ordre

Définition

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

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

Ordre

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**)

Ordre

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**)

Ordre

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 $\mathbb{N}$)

Ordre

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

Ordre

Ordre lexicographique

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

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

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

Ordre

Ordre lexicographique

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

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

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 `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 `ack n p` termine pour tout entiers positifs `n`, `p`.

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

Fonction d'Ackermann

```ocaml
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 `ack n p` termine pour tout entiers positifs `n`, `p`.

Supposons qu'il y ait une infinité d'appels récursifs et appelons (n_k, p_k) les arguments du $k$ème appel récursif.
Clairement, n_k et p_k sont des entiers positifs.
De plus (n_k, p_k) décroit strictement pour $\preceq$: contradiction.

Correction

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
```

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
```

Récurrence 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);;
```

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

Correction

```ocaml
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
```

Correction

```
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 récursifs en OCaml donnent naturellement un schéma de récurrence, appelé **preuve par induction structurelle**.

Induction structurelle

Les types récursifs 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. $\mathcal{P}(\texttt{[]})$
2. $\mathcal{P}(\texttt{l}) \implies \forall\ \texttt{e},\ \mathcal{P}(\texttt{e::l})$

Induction structurelle

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. $\mathcal{P}(\texttt{Vide})$
2. $\mathcal{P}(\texttt{g}) \wedge \mathcal{P}(\texttt{d}) \implies \mathcal{P}(\texttt{Noeud(r, g, d)})$