Plan

Définition

Une fonction d'ordre supérieur est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un paramètre de type fonction ;
2. `f` renvoie une fonction.

Définition

Une fonction d'ordre supérieur est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un paramètre de type fonction ;
2. `f` renvoie une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

Définition

Une fonction d'ordre supérieur est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un paramètre de type fonction ;
2. `f` renvoie une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

Le fait de pouvoir renvoyer une fonction est un procédé très puissant en programmation fonctionnelle. Le programmeur n'est plus le seul concepteur de fonctions : l'exécution/l'interprétation peut en créer à la volée et en appeler.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

`E1 -> E2 -> ... -> En -> S`,

toute application partielle

`(f e1 e2 ... ek)`

avec $1 \leqslant k \leqslant n-1$ produit une valeur de type

`Ek+1 -> ... -> En -> S`

qui est donc une fonction.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

$$\texttt{E1 -> E2 -> ... -> En -> S},$$

toute application partielle

$$\texttt{(f e1 e2 ... ek)}$$

avec $1 \leqslant k \leqslant n-1$ produit une valeur de type

$$\texttt{Ek+1 -> ... -> En -> S}$$

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont curryfiées en Caml : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions**.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

$$\texttt{E1 -> E2 -> ... -> En -> S},$$

toute application partielle

$$\texttt{(f e1 e2 ... ek)}$$

avec $1 \leqslant k \leqslant n-1$ produit une valeur de type

$$\texttt{Ek+1 -> ... -> En -> S}$$

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont curryfiées en Caml : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions**.

On peut donc voir toute fonction à deux paramètres ou plus comme une fonction d'ordre supérieur car son application partielle renvoie une fonction.

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =
    if n = 0 then
        x + 0
    else
        (f (appli_repetee f x (n - 1)));;
```

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =
    if n = 0 then
        x + 0
    else
        (f (appli_repetee f x (n - 1)));;
```

On infère le type

`(int -> int) -> int -> int -> int`,

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =
    if n = 0 then
        x + 0
    else
        (f (appli_repetee f x (n - 1)));;
```

On infère le type

$$\texttt{(int -> int) -> int -> int -> int},$$

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée $\texttt{n}^{\text{e}}$ de `f` sur l'entier `x`, c.-à-d.,

$$(\underbrace{\texttt{f} \circ \cdots \circ \texttt{f}}_{\texttt{n} \text{ fois}})(\texttt{x})$$

Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =
    if n = 0 then
        x + 0
    else
        (f (appli_repetee f x (n - 1)));;
```

On infère le type

$$\texttt{(int -> int) -> int -> int -> int},$$

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée $\texttt{n}^{\text{e}}$ de `f` sur l'entier `x`, c.-à-d.,

$$\underbrace{(\texttt{f} \circ \cdots \circ \texttt{f})}_{\texttt{n} \text{ fois}}(\texttt{x})$$

```
# (appli_repetee (fun x -> 2 * x) 3 4);;
- : int = 48
```

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =
    (fun v -> u ^ v ^ w);;
```

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =
    (fun v -> u ^ v ^ w);;
```

On infère le type

`string -> string -> string -> string`.

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =
    (fun v -> u ^ v ^ w);;
```

On infère le type

`string -> string -> string -> string`.

Il est équivalent (à cause du parenthésage implicite) au type

`string -> string -> (string -> string)`,

Ce qui montre que `encadrer` renvoie une fonction `string -> string`.
C'est donc une fonction d'ordre supérieur.

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =
    (fun v -> u ^ v ^ w);;
```

On infère le type

`string -> string -> string -> string`.

Il est équivalent (à cause du parenthésage implicite) au type

`string -> string -> (string -> string)`,

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =
    (fun v -> u ^ v ^ w);;
```

On infère le type

`string -> string -> string -> string`.

Il est équivalent (à cause du parenthésage implicite) au type

`string -> string -> (string -> string)`,

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = (encadrer "aa" "bb");;
val f : string -> string = <fun>
```

Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =
    (fun v -> u ^ v ^ w);;
```

On infère le type

`string -> string -> string -> string`.

Il est équivalent (à cause du parenthésage implicite) au type

`string -> string -> (string -> string)`,

Ce qui montre que `encadrer` renvoie une fonction `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = (encadrer "aa" "bb");;
val f : string -> string = <fun>
# (f "bab");;
- : string = "aababbb"
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))));;
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))));;
```

On infère le type

$$\texttt{(int -> int) -> (int -> int) -> int -> int},$$

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))));;
```

On infère le type

`(int -> int) -> (int -> int) -> int -> int`,

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

$$\texttt{f}(\texttt{g}(\texttt{f}(\texttt{g}(x)))).$$

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))));;
```

On infère le type

$$\texttt{(int -> int) -> (int -> int) -> int -> int},$$

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

$$\texttt{f}(\texttt{g}(\texttt{f}(\texttt{g}(x)))).$$

```
# let h = (entrelacer (fun x -> x * 2) (fun x -> x + 1));;
val h : int -> int = <fun>
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))));;
```

On infère le type

$$\texttt{(int -> int) -> (int -> int) -> int -> int},$$

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

$$\texttt{f}(\texttt{g}(\texttt{f}(\texttt{g}(x)))).$$

```
# let h = (entrelacer (fun x -> x * 2) (fun x -> x + 1));;
val h : int -> int = <fun>
# (h 3);;
- : int = 18
```

Exemple complet : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers.

Exemple complet : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Exemple complet : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

Exemple complet : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots$$

Exemple complet : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder de manière compacte des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots$$

Question : comment représenter des séries génératrices ?

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Exemple complet : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Par exemple, la série génératrice des puissances de 2 est ainsi codée par

```
let puissances_2 =
    (fun n -> (int_of_float (2. ** (float_of_int n))));;
```

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)+\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k\ ;$$

2. **produit d'Hadamard** :

$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\bullet\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k t^k\ ;$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)+\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k\ ;$$

2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\bullet\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k t^k\ ;$$

3. **produit** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\cdot\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}t^k.$$

Exemple complet : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$

2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \bullet \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \alpha_k \beta_k t^k \ ;$$

3. **produit** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \cdot \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \sum_{i=0}^{k} \alpha_i \beta_{k-i} t^k.$$

Il est possible de les implanter simplement en utilisant les fonctions d'ordre supérieur.

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;
```

```
# let s3 = (somme puissances_2 puissances_2);;
val s3 : int -> int = <fun>
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;
```

```
# let s3 = (somme puissances_2 puissances_2);;
val s3 : int -> int = <fun>
# (s3 3);;
- : int = 16
```

Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res;;
```

```
# let s3 = (somme puissances_2 puissances_2);;
val s3 : int -> int = <fun>
# (s3 3);;
- : int = 16
```

L'implantation du produit d'Hadamard utilise les mêmes idées :

```
let produit_hadamard s1 s2 =
    (fun k -> (s1 k) * (s2 k));;
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;
```

```
# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;
```

```
# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = (produit sg_un sg_un);;
val sg_un_carre : int -> int = <fun>
```

Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then
                0
            else
                (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        (aux 0)
    in
    resultat;;
```

```
# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = (produit sg_un sg_un);;
val sg_un_carre : int -> int = <fun>

# (sg_un_carre 0), (sg_un_carre 1), (sg_un_carre 2), (sg_un_carre 3);;
- : int * int * int * int = (1, 2, 3, 4)
```

Plan

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

- Une **valeur polymorphe** est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé. P.ex.,

```
# Vide;;
- : 'a liste = Vide
```

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« **tout ou un** »).

De cette manière, pour déterminer le type d'un paramètre `x` d'une fonction, le système de typage fonctionne ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
2. si cette étape échoue (ou bien s'il n'y a aucune occurrence de `x`), alors `x` est du type le plus général `'a`.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.
`compare : 'a -> 'a -> int`	Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.
`compare : 'a -> 'a -> int`	Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.
`fst : 'a * 'b -> 'a`	Renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

`( = ) : 'a -> 'a -> bool`	Test d'égalité entre deux valeurs d'un même type.
`( <> ) : 'a -> 'a -> bool`	Test de différence entre deux valeurs d'un même type.
`compare : 'a -> 'a -> int`	Fonction générique de comparaison entre deux valeurs d'un même type. Renvoie `-1` (resp. `1`) si la 1re est strict. inf. (resp. sup.) à la 2e et `0` sinon.
`fst : 'a * 'b -> 'a`	Renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
`snd : 'a * 'b -> 'b`	Renvoie la 2e coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k+1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
    if n = 0 then
        1
    else
        let tmp = (puiss x (n / 2)) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x;;
```

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k+1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
    if n = 0 then
        1
    else
        let tmp = (puiss x (n / 2)) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x;;
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n`.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n`.

Le type de retour est `'e`.

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
# (puiss_poly (fun u v -> u ^ v) "" "abb" 4);;
- : string = "abbabbabbabb"
```

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = (puiss_poly op unite x (n / 2)) in
        if n mod 2 = 0 then
            (op tmp tmp)
        else
            (op (op tmp tmp) x);;
```

Exemples d'utilisation :

```
# (puiss_poly (fun a b -> a + b) 0 1 6);;
- : int = 6
# (puiss_poly (fun a b -> a * b) 1 2 10);;
- : int = 1024
# (puiss_poly (fun u v -> u ^ v) "" "abb" 4);;
- : string = "abbabbabbabb"
# (puiss_poly (fun a b -> a || b) false false 293898273);;
- : bool = false
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

```
'e liste -> 'e -> bool
```

testant la présence de la valeur du 2^{e} paramètre dans le 1er.

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

`'e liste -> 'e -> bool`

testant la présence de la valeur du 2e paramètre dans le 1er.

```
let rec appartient_liste lst x =
    match lst with
        |Vide -> false
        |(Cellule (y, _)) when y = x -> true
        |(Cellule (_, reste)) -> (appartient_liste reste x);;
```

Exemple : listes génériques — appartenance

Soit le type

```
type 'e liste = Vide | Cellule of 'e * 'e liste;;
```

On souhaite écrire une fonction polymorphe `appartient_liste` de type

`'e liste -> 'e -> bool`

testant la présence de la valeur du 2e paramètre dans le 1er.

```
let rec appartient_liste lst x =
    match lst with
        |Vide -> false
        |(Cellule (y, _)) when y = x -> true
        |(Cellule (_, reste)) -> (appartient_liste reste x);;
```

Ceci se base sur le fait que test d'égalité `=` est polymorphe (de type `'a -> 'a -> 'a` ; il n'est donc pas à fournir en paramètre à la fonction).

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

Exemple : listes génériques — maximum

On souhaite maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le plus grand élément d'une liste générique. Celle-ci est de type

```
('e -> 'e -> 'e) -> 'e liste -> 'e
```

où

- le 1er paramètre est une fonction qui prend comme arguments deux éléments et renvoie le plus grand, codant une relation d'ordre totale.
- Le 2e paramètre est la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est de type `'e`.

```
let maximum_liste mmax lst =
    let rec aux lst max_prefixe =
        match lst with
            |Vide -> max_prefixe
            |(Cellule (x, reste)) -> (aux reste (mmax max_prefixe x))
    in
    match lst with
        |Vide -> (failwith "liste vide")
        |(Cellule (x, reste)) -> (aux reste x);;
```

Plan

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une non-terminaison ;

Exemple introductif

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la stratégie d'évaluation du langage, c.-à-d., de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)`.

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une non-terminaison ;
2. sinon, l'expression `f(-1)` n'est pas évaluée car le second argument, `0`, de l'appel à `g` est pair. L'exécution termine dans ce cas.

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1)  (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1)  (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
            ⇝ (f 1 (f 2 1 4) 12)
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1)  (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
            ⇝ (f 1 (f 2 1 4) 12)
            ⇝ (f 1 6 12)
```

Appel par valeur

La stratégie d'évaluation en appel par valeur consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1)  (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
            ⇝ (f 1 (f 2 1 4) 12)
            ⇝ (f 1 6 12)
            ⇝ 13
```

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;
2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Appel par valeur — inconvénients

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;
2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Beaucoup de langages utilisent cette stratégie, dont le Caml.

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
```

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (1 * 1) + (3 * 4)
```

Appel par nom

La stratégie d'évaluation en appel par nom consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

P.ex., si l'on a une fonction

```
let f x y z =
    x + z;;
```

l'expression

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4));;
```

s'évalue au moyen des étapes suivantes :

```
(f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4))
          ⇝ (1 * 1) + (3 * 4)
          ⇝ 13
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3)  (2 * 1))
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3)  (2 * 1))
```

$\rightsquigarrow$ `(4 * 3) * (4 * 3) + (2 * 1)`

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3)  (2 * 1))
          ⇝ (4 * 3) * (4 * 3) + (2 * 1)
          ⇝ 146
```

Appel par nom — inconvénients

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

En effet, si l'on a une fonction

```
let f x y =
    x * x + y;;
```

l'expression

```
(f (4 * 3) (2 * 1));;
```

s'évalue en appel par nom au moyen des étapes suivantes :

```
(f (4 * 3)  (2 * 1))
          ⇝ (4 * 3) * (4 * 3) + (2 * 1)
          ⇝ 146
```

L'argument `(4 * 3)` est évalué ainsi deux fois (au lieu d'une seule que ferait un appel par valeur).

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

Appel par nécessité

La stratégie en appel par nécessité est une **version mémoïsée de l'appel par nom**.

Lors de l'application d'une fonction `f` à des expressions `a1`, . . ., `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un `ai`, si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de `f` correspondant à un `ai` ne redemande pas d'être réévaluée.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le principe de transparence référentielle est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le principe de transparence référentielle est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Appel par nécessité

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le principe de transparence référentielle est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Le langage Haskell utilise cette stratégie.