PF chapître 4 : typage et ordre supérieur

Jean-François Monin

POLYTECH GRENOBLE

Plan

Plan

Fonctions à 1 argument

Mathématiques : $x \mapsto f(x)$

OCaml : `fun x -> f x`

Fonctions à 1 argument

Mathématiques : $x \mapsto f(x)$

OCaml : `fun x -> f x`

Exemple : prédicat « est non nul »

Fonctions à 1 argument

Mathématiques : $x \mapsto f(x)$ OCaml : `fun x -> f x`

Exemple : prédicat « est non nul »

```
# let estnonnul = fun x -> x <> 0 ;;
val estnonnul : int -> bool = <fun>
```

(La fonction qui, à tout entier x, associe la valeur de « $x \neq 0$ »)

Fonctions à 1 argument

Mathématiques : $x \mapsto f(x)$ OCaml : `fun x -> f x`

Exemple : prédicat « est non nul »

```
# let estnonnul = fun x -> x <> 0 ;;
val estnonnul : int -> bool = <fun>
```

(La fonction qui, à tout entier x, associe la valeur de « $x \neq 0$ »)

Définition abrégée

```
# let estnonnul x = x <> 0 ;;
```

Fonctions à plusieurs arguments

Point de vue au premier ordre (usuel en maths)

fonction à n arguments = fonction à 1 argument : n-uplet

Norme $\sqrt{x^2 + y^2 + z^2}$ d'un vecteur

norme : `float` $\times$ `float` $\times$ `float` $\rightarrow$ `float`

```
# let norme (x, y, z) = sqrt (x *. x +. y *. y +. z *.z)
```

Fonctions à plusieurs arguments

Point de vue au premier ordre (usuel en maths)

fonction à n arguments = fonction à 1 argument : n-uplet

Norme $\sqrt{x^2 + y^2 + z^2}$ d'un vecteur

norme : `float` × `float` × `float` → `float`

```
# let norme (x, y, z) = sqrt (x *. x +. y *. y +. z *.z)
```

Point de vue à l'ordre supérieur (préféré en prog. fonctionnelle)

- Fonction à 0 argument = constante
- Fonction à $n + 1$ arguments =
  fonction à 1 argument qui rend une fonction à n arguments

Fonctions à plusieurs argument

Curryfication

Fonction à $n+1$ arguments $=$
fonction à 1 argument qui rend une fonction à n arguments

Fonctions à plusieurs argument

Curryfication

Fonction à $n+1$ arguments $=$
fonction à 1 argument qui rend une fonction à n arguments

```
# let plus = fun x -> (fun y -> x + y)
plus : int -> (int -> int)
```

Fonctions à plusieurs argument

Curryfication

Fonction à $n+1$ arguments =
fonction à 1 argument qui rend une fonction à n arguments

```
# let plus = fun x -> (fun y -> x + y)
plus : int -> (int -> int)
```

```
# let plus = fun x -> fun y -> x + y
plus : int -> int -> int
```

Fonctions à plusieurs argument

Curryfication

Fonction à $n+1$ arguments =
fonction à 1 argument qui rend une fonction à n arguments

```
# let plus = fun x -> (fun y -> x + y)
plus : int -> (int -> int)
```

```
# let plus = fun x -> fun y -> x + y
plus : int -> int -> int
```

`plus 3 2` se lit `(plus 3) 2`

Fonctions à plusieurs argument

Curryfication

Fonction à $n+1$ arguments =
fonction à 1 argument qui rend une fonction à n arguments

```
# let plus = fun x -> (fun y -> x + y)
plus : int -> (int -> int)
```

```
# let plus = fun x -> fun y -> x + y
plus : int -> int -> int
```

`plus 3 2` se lit `(plus 3) 2`

```
# let norme x y z = sqrt (x *. x +. y *. y +. z *.z)
norme : float -> float -> float -> float
norme : float -> (float -> (float -> float))
```

Curryfication de la fonction d'addition

$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$

```
plusA (3,4)
```

$\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$

```
plusB 3 2      se lit      (plusB 3) 2
```

plusB : $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$ qui se lit $\texttt{int} \rightarrow (\texttt{int} \rightarrow \texttt{int})$

Curryfication de la fonction d'addition

$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$

```
plusA (3,4)
```

$\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$

`plusB 3 2` se lit `(plusB 3) 2`

plusB : $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$ qui se lit $\texttt{int} \rightarrow (\texttt{int} \rightarrow \texttt{int})$

Que rend `plusB 3` ?

Curryfication de la fonction d'addition

$\texttt{int} \times \texttt{int} \to \texttt{int}$

```
plusA (3,4)
```

$\texttt{int} \to \texttt{int} \to \texttt{int}$

```
plusB 3 2      se lit      (plusB 3) 2
```

plusB : $\texttt{int} \to \texttt{int} \to \texttt{int}$ qui se lit $\texttt{int} \to (\texttt{int} \to \texttt{int})$

Que rend `plusB 3` ?

Que rend `plusA 3` ?

Abréviations pour les définitions de fonctions

```
let somme3 = fun x -> (fun y -> (fun z -> x + y + z))
let somme3 = fun x y z -> x + y + z
let somme3 x y z = x + y + z
```

Plan

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple

```
# let aff23 = fun x -> 2 * x + 3
```

Comment obtenir le résultat rendu par `aff23` en 1 ?

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple

```
# let aff23 = fun x -> 2 * x + 3
```

Comment obtenir le résultat rendu par `aff23` en 1 ?

```
aff23 1
```

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple
let `aff23` = `fun` $x \rightarrow 2 * x + 3$

Comment obtenir le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire f sur les entiers

Comment obtenir le résultat rendu par f en 1 ?

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple

let `aff23` = `fun` $x \rightarrow 2 * x + 3$

Comment obtenir le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire f sur les entiers

Comment obtenir le résultat rendu par f en 1 ?

$f\,1$

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple
let `aff23` = `fun` $x \rightarrow 2 * x + 3$

Comment obtenir le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire f sur les entiers

Comment obtenir le résultat rendu par f en 1 ?

$f\,1$

Fonction `val1` qui rend la valeur en 1 d'une fonction f

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple

let `aff23` = `fun` $x \rightarrow 2 * x + 3$

Comment obtenir le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire f sur les entiers

Comment obtenir le résultat rendu par f en 1 ?

$f\,1$

Fonction `val1` qui rend la valeur en 1 d'une fonction f

let `val1` = `fun` $f \rightarrow f\ 1$

Les fonctions peuvent être en argument d'une fonction

On considère une fonction sur les entiers à 1 argument, exemple
let `aff23` = `fun` $x \rightarrow 2 * x + 3$

Comment obtenir le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire f sur les entiers

Comment obtenir le résultat rendu par f en 1 ?

$f\,1$

Fonction `val1` qui rend la valeur en 1 d'une fonction f
let `val1` = `fun` $f \rightarrow f\ 1$

Que rend `val1 aff23` ?

Problème

Écrire une fonction `affine` prenant deux entiers a et b, et renvoyant la fonction $x \to ax + b$?

Problème

Écrire une fonction `affine` prenant deux entiers a et b, et renvoyant la fonction $x \to ax + b$?

- Dans la plupart des langages traditionnels : exercice difficile
- Pas de *valeur* de type : "fonction int vers int"

Problème

Écrire une fonction `affine` prenant deux entiers a et b, et renvoyant la fonction $x \to ax + b$?

- Dans la plupart des langages traditionnels : exercice difficile
- Pas de *valeur* de type : "fonction int vers int"
- En OCaml les fonctions sont *traitées comme des données*

Les fonctions peuvent être en résultat d'une fonction

Déjà vu

```
# let plus = fun x -> (fun y -> x + y)
```

Fonction affine

```
# let affine = fun a -> fun b -> (fun x -> a * x + b)
```

Les fonctions peuvent être en résultat d'une fonction

Déjà vu

```
# let plus = fun x -> (fun y -> x + y)
```

Fonction affine

```
# let affine = fun a -> fun b -> (fun x -> a * x + b)
```

Qu'est ce que (`affine 2`) ?

Les fonctions peuvent être en résultat d'une fonction

Déjà vu

```
# let plus = fun x -> (fun y -> x + y)
```

Fonction affine

```
# let affine = fun a -> fun b -> (fun x -> a * x + b)
```

Qu'est ce que (`affine 2`) ?

Qu'est ce que ((`affine 2`) 3) ?

Les fonctions peuvent être en résultat d'une fonction

Déjà vu

```
# let plus = fun x -> (fun y -> x + y)
```

Fonction affine

```
# let affine = fun a -> fun b -> (fun x -> a * x + b)
```

Qu'est ce que `(affine 2)` ?

Qu'est ce que `((affine 2) 3)` ?

Que rend `val1` `((affine 2) 3)` ?

Fonctions en argument et résultat d'une fonction

Exemple : Composition

```
# let compo f g = fun x → f (g x)
val compo : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Fonctions en argument et résultat d'une fonction

Exemple : Composition

```
# let compo f g = fun x → f (g x)
val compo : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let f = compo (fun y → (y, y + 1))(fun t → t * 4)
val f : int -> int * int = <fun>

# let _ = f 1
```

Fonctions en argument et résultat d'une fonction

Exemple : Composition

```
# let compo f g = fun x → f (g x)
val compo : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let f = compo (fun y → (y, y + 1))(fun t → t * 4)
val f : int -> int * int = <fun>

# let _ = f 1
- : int * int = (4, 5)
```

Plan

Plan

Système de types

- Types de données
  - simples : numériques, booléens, chaînes
  - structurés : types produits, types sommes
- Types fonctionnels
  - fonction = valeur ordinaire

Toutes les combinaisons sont permises, par ex. listes de fonctions

Généralités sur le typage statique

Objectif : rejet de programme absurdes

$(1\ 2)$ ou $(\mathit{fun}\ x\ \rightarrow x) + 1$

Propriétés attendues pour un système de types

- Décidable
- Correct
- Expressif

Généralités sur le typage statique

Objectif : rejet de programme absurdes

$(1\ 2)$ ou $(\mathit{fun}\ x\ \rightarrow x) + 1$

Propriétés attendues pour un système de types

- Décidable
- Correct
- Expressif
- Confortable

Des exemples simples

fun x $\rightarrow$ x

Des exemples simples

fun x → x

int → int
bool → bool
$\alpha \to \alpha$

fun x → x + 1

Des exemples simples

fun x → x

int → int
bool → bool
$\alpha \to \alpha$

fun x → x + 1

int → int
mais pas
bool → int

1 2

Des exemples simples

fun x → x

int → int
bool → bool
$\alpha \to \alpha$

fun x → x + 1

int → int
mais pas
bool → int

1 2

Pas typable, car 1 n'est pas de type int → 'a

Un exemple moins simple

$$\text{fun } x \rightarrow x\ x$$

Un exemple moins simple

fun x $\rightarrow$ x x

Pas typable en OCaml, car x de type $\alpha \rightarrow \beta$ et α en même temps

Un exemple moins simple

fun x → x x

Pas typable en OCaml, car x de type $\alpha \to \beta$ et α en même temps

Et dans un système de types plus riche ?

- Possible (système F de Girard-Reynols), mais peu utilisé en programmation
- Demande des indications explicites de typage

En OCaml

En OCaml les types sont inférés automatiquement.

Pour les données

- Types de base : `int`, `float`, `char`, `string`, `bool`, `unit`
- Types construits : produits (juxtaposition) et sommes (choix)

Pour les fonctions

Exemple : `fun` x $\rightarrow$ 1 + x : `int` $\rightarrow$ `int`

Types polymorphes (comparables à la généricité en Ada)

- `longueur` : α `list` $\rightarrow$ `int`
- `fun` x $\rightarrow$ (x, x) : $\alpha \rightarrow \alpha \times \alpha$

Typage dans un environnement

$$\boxed{\Gamma \vdash \mathit{expression} : \mathit{type}}$$

Γ *environnement de typage* = liste d'associations $x : t$,
où x est un nom et t un type

Exemples

- $\mathit{nbr} : \texttt{int} \vdash \mathit{nbr} + 4 : \texttt{int}$
- $\mathit{nbr} : \texttt{int} ; \mathit{b1} : \texttt{bool} \vdash \mathit{nbr} + 4 : \texttt{int}$
- $\mathit{nbr} : \texttt{int}; \mathit{b1} : \texttt{bool} ; \mathit{nbr} : \texttt{float} \vdash \mathit{nbr} +. 3.14 : \texttt{float}$

Règles initiales de typage

Constantes littérales

$$\frac{}{\Gamma \vdash 0 : \texttt{int}}$$

Et similairement pour $\ldots, -2, -1, 1, 2, \ldots$, les flottants, les caractères, les chaînes, etc.

Gestion de l'environnement de typage

$$\frac{}{\Gamma; x : t \vdash x : t}$$

$$\frac{\Gamma \vdash x : t \qquad x \neq y}{\Gamma; y : u \vdash x : t}$$

NB. Environnement parcouru à partir de la droite

Expression conditionnelle

Règle de typage

$$\frac{\Gamma \vdash B : \texttt{bool} \qquad \Gamma \vdash E_1 : t \qquad \Gamma \vdash E_2 : t}{\Gamma \vdash (\texttt{if } B \texttt{ then } E_1 \texttt{ else } E_2) : t}$$

Remarques

- Type calculé à la fois pour E_1 et E_2 *($\neq$ valeur)*
- Différence entre le typage *statique* et l'évaluation *dynamique* : système de types décidable

Typage des couples et n-uplets

Construction

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

Similairement pour des n-uplets

Décomposition

Par filtrage, cf. plus bas.

Fonctions

A, B et E sont des expressions

Notation Ocaml : `fun` x $\rightarrow$ E

Règles de typage

$$\frac{\Gamma \vdash A : t \rightarrow u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\,B : u}$$

$$\frac{\Gamma ; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\, x \rightarrow E : t \rightarrow u}$$

Typage d'une expression

L'environnement Γ_0 initialement chargé (appelé *Stdlib*) contient :

- $(+)$: `int` $\rightarrow$ `int` $\rightarrow$ `int` ; etc.
- $(+.)$: `float` $\rightarrow$ `float` $\rightarrow$ `float` ; etc.
- $(\&\&)$: `bool` $\rightarrow$ `bool` $\rightarrow$ `bool` ; etc.

Typage d'une expression

L'environnement Γ_0 initialement chargé (appelé *Stdlib*) contient :

- $(+) : \texttt{int} \to \texttt{int} \to \texttt{int}$; etc.
- $(+.) : \texttt{float} \to \texttt{float} \to \texttt{float}$; etc.
- $(\&\&) : \texttt{bool} \to \texttt{bool} \to \texttt{bool}$; etc.

$$\dfrac{\dfrac{\Gamma_0 \vdash (+) : \texttt{int} \to \texttt{int} \to \texttt{int} \qquad \Gamma_0 \vdash 3 : \texttt{int}}{\Gamma_0 \vdash (+)\,3 : \texttt{int} \to \texttt{int}} \qquad \Gamma_0 \vdash 2 : \texttt{int}}{\Gamma_0 \vdash (+)\,3\,2 : \texttt{int}}$$

Exemple

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\, x \to E : t \to u}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\, B : u}$$

Vérifier que :

$\Gamma_0 \vdash \texttt{fun}\, x \to x + 1 : \texttt{int} \to \texttt{int}$

Exemple

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\, x \to E : t \to u}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\, B : u}$$

Vérifier que :

$\Gamma_0 \vdash \texttt{fun}\, x \to x + 1 : \texttt{int} \to \texttt{int}$

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash + : int \to int \to int \qquad \Gamma_1 \vdash x : int}{\Gamma_1 \vdash (+)\, x : int \to int} \qquad \dfrac{}{\Gamma_1 \vdash 1 : int}}{\Gamma_1 = \Gamma_0; x : int \vdash x + 1 : int}}{\Gamma_0 \vdash fun\ x \to x + 1 : int \to int}$$

Exercice

Sachant que l'environnement Γ_0 contient $\texttt{s} : \texttt{int} \to \texttt{int}$
(et aucun autre couple $\texttt{s} : t$)
vérifier que $\Gamma_0 \vdash \texttt{fun}\ n \to \texttt{s}\ (\texttt{s}\ n) : \texttt{int} \to \texttt{int}$

Notation (pour raison de place)

- $\Gamma_1 \stackrel{\text{déf}}{=\!=} \Gamma_0; n : \texttt{int}$

Exercice

Sachant que l'environnement Γ_0 contient $\texttt{s} : \texttt{int} \to \texttt{int}$
(et aucun autre couple $\texttt{s} : t$)
vérifier que $\Gamma_0 \vdash \texttt{fun}\ n \to \texttt{s}\ (\texttt{s}\ n) : \texttt{int} \to \texttt{int}$

Notation (pour raison de place)

- $\Gamma_1 \stackrel{\text{déf}}{=\!=} \Gamma_0; n : \texttt{int}$

$$\dfrac{\dfrac{\Gamma_1 \vdash \texttt{s} : \texttt{int} \to \texttt{int} \qquad \dfrac{}{\Gamma_0; n : \texttt{int} \vdash n : \texttt{int}}}{\Gamma_1 \vdash \texttt{s} : \texttt{int} \to \texttt{int} \qquad \Gamma_0; n : \texttt{int} \vdash \texttt{s}\ n : \texttt{int}}}{\dfrac{\Gamma_1 = \Gamma_0; n : \texttt{int} \vdash \texttt{s}\ (\texttt{s}\ n) : \texttt{int}}{\Gamma_0 \vdash \texttt{fun}\ n \to \texttt{s}\ (\texttt{s}\ n) : \texttt{int} \to \texttt{int}}}$$

Plan

Typage du let

$$\frac{\Gamma \vdash A : t \qquad \Gamma; x : t \vdash B : u}{\Gamma \vdash \textbf{let}\ x = A\ \texttt{in}\ B\ :\ u}$$

Typage du let

$$\frac{\Gamma \vdash A : t \qquad \Gamma; x : t \vdash B : u}{\Gamma \vdash \textbf{let } x = A \texttt{ in } B : u}$$

Exemple

```
# let x = 5 in x ;;
- : int = 5
```

$$\frac{\Gamma \vdash 5 : int \qquad \Gamma; x : int \vdash x : int}{\Gamma \vdash \textbf{let } x = 5 \texttt{ in } x : int}$$

Exercice

$$\frac{\Gamma \vdash A : t \qquad \Gamma, x : t \vdash B : u}{\Gamma \vdash \textbf{let } x = A \texttt{ in } B \ : \ u}$$

Vérifier le type de :

```
# let succ = fun x → x + 1 in succ 1 ;;
- : int = 2
val x : int → int = <fun>
```

Exercice

$$\frac{\Gamma \vdash A : t \qquad \Gamma, x : t \vdash B : u}{\Gamma \vdash \textbf{let } x = A \texttt{ in } B \ : \ u}$$

Vérifier le type de :

```
# let succ = fun x -> x + 1 in succ 1 ;;
- : int = 2
val x : int -> int = <fun>
```

$$\frac{\dfrac{\vdots}{\Gamma \vdash \mathit{fun}\ x \to x+1 : \mathit{int} \to \mathit{int}} \qquad \dfrac{\Gamma_0 \vdash \mathit{succ} : \mathit{int} \to \mathit{int} \qquad \Gamma_0 \vdash 1 : \mathit{int}}{\Gamma_0 = \Gamma; \mathit{succ} : \mathit{int} \to \mathit{int} \vdash \mathit{succ}\ 1 : \mathit{int}}}{\Gamma \vdash \textbf{let } \mathit{succ} = \mathit{fun}\ x \to x+1 \texttt{ in } \mathit{succ}\ 1 \ : \ \mathit{int}}$$

Typage du let rec

$$\frac{\Gamma; x : t \vdash A : t \qquad \Gamma; x : t \vdash B : u}{\Gamma \vdash \textbf{let rec } x = A \texttt{ in } B : u}$$

Typage du let rec

$$\frac{\Gamma; x : t \vdash A : t \qquad \Gamma; x : t \vdash B : u}{\Gamma \vdash \textbf{let rec } x = A \texttt{ in } B : u}$$

Exemple : fact

```
# let rec fact = fun n → if n = 0 then 1 else n * fact (n-1) ;;
val fact : int → int = <fun>
```

Au tableau

Typage du filtrage (match)

type $t = C_1$ of $t_1 \mid C_2$ of $u_1 \times u_2 \mid \ldots$

Posons

$$M = \begin{cases} \texttt{match } E \texttt{ with} \\ \mid C_1(x_1) \rightarrow E_1 \\ \mid C_2(y_1, y_2) \rightarrow E_2 \\ \vdots \end{cases}$$

$$\frac{\Gamma \vdash E : t \qquad \Gamma; x_1 : t_1 \vdash E_1 : u \qquad \Gamma; y_1 : u_1, y_2 : u_2 \vdash E_2 : u \ldots}{\Gamma \vdash M : u}$$

Typage du match

Exemple : len

```
# let rec len = fun l -> match l with
      | Nil -> 0
      | Cons(h, t) -> 1 + (len t) ;;
val len : 'a list -> int = <fun>
```

A faire en utilisant le typage de let rec et du match

En particulier on est amené à montrer

$\Gamma, \mathit{len} : {}'a\ \text{list} \to \mathit{int}, \mathit{h} : {}'a, \mathit{t} : {}'a\ \text{list} \vdash 1 + (\text{len t}) : \mathit{int}$

Plan

En somme, qu'est-ce qu'un type ?

Moyens de construction élémentaires : (principes *d'introduction*)

Indiquent les valeurs du type (non réductibles)

- produits
- sommes
- exponentielles (fonctions, tableaux)

En somme, qu'est-ce qu'un type ?

Moyens de construction élémentaires : (principes *d'introduction*)

Indiquent les valeurs du type (non réductibles)

- produits
- sommes
- exponentielles (fonctions, tableaux)

Procédés d'utilisation élémentaires : (principes *d'élimination*)

- projection **let** (x,y) = ... (produits)
- filtrage (sommes)
- application (fonctions), accès (tableaux)

En somme, qu'est-ce qu'un type ?

Moyens de construction élémentaires : (principes *d'introduction*)

Indiquent les valeurs du type (non réductibles)

- produits
- sommes
- exponentielles (fonctions, tableaux)

Procédés d'utilisation élémentaires : (principes *d'élimination*)

- projection **let** (x,y) = ... (produits)
- filtrage (sommes)
- application (fonctions), accès (tableaux)

Symétrie introduction / élimination

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash \qquad t \qquad \Gamma \vdash \qquad u}{\Gamma \vdash \qquad t * u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t * u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t * u}{\Gamma \vdash t}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t \wedge u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t * u}{\Gamma \vdash t}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t \wedge u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t \wedge u}{\Gamma \vdash t}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t \wedge u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t \wedge u}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\ B : u}$$

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\ x \to E : t \to u}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t \wedge u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t \wedge u}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\ B : u}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\ B : u}$$

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\ x \to E : t \to u}$$

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\ x \to E : t \to u}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t \wedge u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t \wedge u}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\ B : u}$$

$$\frac{\Gamma \vdash t \to u \qquad \Gamma \vdash t}{\Gamma \vdash u}$$

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\ x \to E : t \to u}$$

$$\frac{\Gamma; t \vdash u}{\Gamma \vdash t \to u}$$

Preuve et programme, même combat

$$\frac{\Gamma \vdash A : t \qquad \Gamma \vdash B : u}{\Gamma \vdash (A, B) : t * u}$$

$$\frac{\Gamma \vdash C : t * u}{\Gamma \vdash \textbf{let}\ (x, \_) = C\ \texttt{in}\ x : t}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash u}{\Gamma \vdash t \wedge u}$$

$$\frac{\Gamma \vdash t \wedge u}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash A : t \to u \qquad \Gamma \vdash B : t}{\Gamma \vdash A\ B : u}$$

$$\frac{\Gamma; x : t \vdash E : u}{\Gamma \vdash \texttt{fun}\ x \to E : t \to u}$$

$$\frac{\Gamma \vdash t \to u \qquad \Gamma \vdash t}{\Gamma \vdash u}$$

$$\frac{\Gamma; t \vdash u}{\Gamma \vdash t \to u}$$

Système de typage (PF)

Système de déduction (LT)

Que fait-on d'un système de types ?

- Vérification pure (comme partout)
  fun (x :int) → (+ : int → int → int) (3 :int) (x :int) : int

Que fait-on d'un système de types ?

- Vérification pure (comme partout)
  fun (x :int) $\rightarrow$ (+ : int $\rightarrow$ int $\rightarrow$ int) (3 :int) (x :int) : int
- Déclaration des fonctions et variables locales et propagation des types
  fun (x :int) $\rightarrow$ let (y : int) = 3 in (+) y x

Que fait-on d'un système de types ?

- Vérification pure (comme partout)
  fun (x :int) $\rightarrow$ (+ : int $\rightarrow$ int $\rightarrow$ int) (3 :int) (x :int) : int
- Déclaration des fonctions et variables locales et propagation des types
  fun (x :int) $\rightarrow$ let (y : int) = 3 in (+) y x
- Déclaration des fonctions et propagation des types
  fun (x :int) $\rightarrow$ let y = 3 in (+) y x

Que fait-on d'un système de types ?

- Vérification pure (comme partout)
  fun (x :int) $\rightarrow$ (+ : int $\rightarrow$ int $\rightarrow$ int) (3 :int) (x :int) : int
- Déclaration des fonctions et variables locales et propagation des types
  fun (x :int) $\rightarrow$ let (y : int) = 3 in (+) y x
- Déclaration des fonctions et propagation des types
  fun (x :int) $\rightarrow$ let y = 3 in (+) y x
- Inférence complète
  fun x $\rightarrow$ let y = 3 in (+) y x

Que fait-on d'un système de types ?

- Vérification pure (comme partout)
  fun (x :int) $\rightarrow$ (+ : int $\rightarrow$ int $\rightarrow$ int) (3 :int) (x :int) : int
- Déclaration des fonctions et variables locales et propagation des types
  fun (x :int) $\rightarrow$ let (y : int) = 3 in (+) y x
- Déclaration des fonctions et propagation des types
  fun (x :int) $\rightarrow$ let y = 3 in (+) y x
- Inférence complète
  fun x $\rightarrow$ let y = 3 in (+) y x

Propriétés

- Correction (*si* succès de l'inférence *alors* terme typable)
- Complétude (*si* terme typable *alors* succès de l'inférence)

Inférence de types

Principe

Soit une application $f\,k$

- inférer le type de k : a
- inférer le type de f : nécessairement de la forme $b \to c$
- vérifier que $a = b$
- le type inféré pour $f\,k$ est c

Plan

Le polymorphisme : définitions

Définitions

- Étymologiquement :

Le polymorphisme : définitions

Définitions

- Étymologiquement : plusieurs formes

Le polymorphisme : définitions

Définitions

- Étymologiquement : plusieurs formes
- En informatique :

Le polymorphisme : définitions

Définitions

- Étymologiquement : plusieurs formes
- En informatique : capacité d'une fonction à « s'adapter »à des arguments de type différent

Le polymorphisme : définitions

Définitions

- Étymologiquement : plusieurs formes
- En informatique : capacité d'une fonction à « s'adapter »à des arguments de type différent

Deux espèces (hors langages à objets) :

- polymorphisme *ad-hoc*
  ex : $+$ sur les entiers, les flottants, les chaînes...
  dans chaque type, le code exécuté est différent

Le polymorphisme : définitions

Définitions

- Étymologiquement : plusieurs formes
- En informatique : capacité d'une fonction à « s'adapter »à des arguments de type différent

Deux espèces (hors langages à objets) :

- polymorphisme *ad-hoc*
  ex : $+$ sur les entiers, les flottants, les chaînes...
  dans chaque type, le code exécuté est différent
- polymorphisme *paramétrique*
  ex : @ sur les listes d'entiers, de flottants, de chaînes...
  le code exécuté est uniformément le même

Exemples basiques *

Fonction identité **let** id = `fun` x → *x*

type :

Exemples basiques *

Fonction identité **let** id = `fun` x → x

type :

$\alpha \to \alpha$

Application :

- à des entiers : id 3
- à des arbres : id (N (N (F, 7, F), 2, N (F, 5, F)))
- à des fonctions : id (`fun` n → (n +3))
- à elle-même : id id, id id 3.14

Exemples basiques *

Fonction identité **let** id = `fun` x → *x*

type :

$\alpha \to \alpha$

Application :

- à des entiers : id 3
- à des arbres : id (N (N (F, 7, F), 2, N (F, 5, F)))
- à des fonctions : id (`fun` n → (*n* +3))
- à elle-même : id id, id id 3.14

Couple

let pairing = `fun` x y → (x,y) ;;

Exemples basiques *

Fonction identité **let** id = `fun` x → *x*

type :

$\alpha \to \alpha$

Application :

- à des entiers : id 3
- à des arbres : id (N (N (F, 7, F), 2, N (F, 5, F)))
- à des fonctions : id (`fun` n → (*n* +3))
- à elle-même : id id, id id 3.14

Couple

let pairing = `fun` x y → (x,y) ;;

```
val pairing : 'a → 'b → 'a * 'b = <fun>
```

Le polymorphisme dans les données (1)

Listes

type α liste = Nil | Cons of $\alpha \times \alpha$ liste

Application (avec la notation OCaml)

- à des entiers : [3 ; 0 ; 8]
- à des arbres : [N (F, 7, F) ; F ; N (F, 5, F)]
- à des fonctions : [fun n → (*n* +3) ; (∗) 6 ; fact]
- à des fonctions polymorphes :
  [(fun x → *x*) ; (fun x → raise Exc)]
- à des listes : [[1 ; 2 ; 3] ; [1 ; 5] ; [] ; [7 ; 6 ; 2]]]

Attention à l'uniformité !

Le polymorphisme dans les données (2)

Type option (en standard)

type α option = None | Some of α

Technique possible pour les fonctions partielles

```
let tete = fun l ->
    match l with
    |   []   -> None
    | x :: _ -> Some (x)
```

Type :

tete : α list $\rightarrow$ α option

Le polymorphisme ad-hoc en Ocaml

Sur les opérateurs de comparaison : $=$, $<$, $<=$, ...

$\alpha \to \alpha \to \texttt{bool}$

Disponible sur toutes les structures de données

- entiers : $14 < 5 + 5$
  et caractères, booléens, chaînes
- listes : $[2\,;\,5\,;\,7\,] < [3\,;\,4\,]$
  et arbres, structures récursives...

Le polymorphisme ad-hoc en Ocaml

Sur les opérateurs de comparaison : $=$, $<$, $<=$, ...

$\alpha \to \alpha \to \texttt{bool}$

Disponible sur toutes les structures de données

- entiers : $14 < 5 + 5$
  et caractères, booléens, chaînes
- listes : $[2\,;\,5\,;\,7\,] < [3\,;\,4\,]$
  et arbres, structures récursives...

Mais pas sur les types fonctionnels

Inférence de types et polymorphisme

Principe

Soit une application $f\,k$

- inférer le type de k : a
- inférer le type de f : nécessairement de la forme $b \to c$
- *trouver la substitution* σ *la plus générale* telle que $\sigma a = \sigma b$ (unificateur)
- le type inféré pour $f\,k$ est σc

Unification de types, exemple

unifier $\alpha \to \alpha$ et $\mathit{int} \to \beta$?

Unification de types, exemple

unifier $\alpha \to \alpha$ et $\mathit{int} \to \beta$?

$\alpha = \mathit{int}$ et $\alpha = \beta$
donc : remplacer α par *int* ; remplacer β par *int*

Un tel procédé peut être réalisé par un **algorithme** correct, complet et qui donne l'unificateur le plus général.

Inférence de types : exemples

- $(\texttt{fun}\ x \rightarrow\ x)\ 3 : ?$

Inférence de types : exemples

- $(\texttt{fun}\ x \rightarrow\ x)\ 3 : ?$
  $\texttt{fun}\ x \rightarrow\ x : \alpha \rightarrow \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$

Inférence de types : exemples

- $(\texttt{fun}\ x \rightarrow\ x)\ 3 : ?$
  $\texttt{fun}\ x \rightarrow\ x : \alpha \rightarrow \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$
  mais $\alpha = \texttt{int}$

Inférence de types : exemples

- $(\texttt{fun}\ x \to\ x)\ 3 : \texttt{int}$
  $\texttt{fun}\ x \to\ x : \alpha \to \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$
  mais $\alpha = \texttt{int}$

Inférence de types : exemples

- $(\texttt{fun}\ x \to\ x)\ 3 : \texttt{int}$
  $\texttt{fun}\ x \to\ x : \alpha \to \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$
  mais $\alpha = \texttt{int}$
- **let** $f = \texttt{fun}\ (x, y) \to\ x + y$: ?

Inférence de types : exemples

- $(\texttt{fun}\ x \to\ x)\ 3 : \texttt{int}$
  $\texttt{fun}\ x \to\ x : \alpha \to \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$
  mais $\alpha = \texttt{int}$
- **let** $f = \texttt{fun}\ (x, y) \to\ x + y\ :?$
  $f : \alpha \times \beta \to \gamma$

Inférence de types : exemples

- $(\texttt{fun}\ x \to\ x)\ 3 : \texttt{int}$
  $\texttt{fun}\ x \to\ x : \alpha \to \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$
  mais $\alpha = \texttt{int}$

- **let** $f = \texttt{fun}\ (x, y) \to\ x + y\ :?$
  $f : \alpha \times \beta \to \gamma$
  $(+) : \texttt{int} \to \texttt{int} \to \texttt{int}$
  d'où $\alpha = \texttt{int}$ et $\beta = \texttt{int}$ et $\gamma = \texttt{int}$

Inférence de types : exemples

- $(\texttt{fun}\ x \to\ x)\ 3 : \texttt{int}$
  $\texttt{fun}\ x \to\ x : \alpha \to \alpha$
  $3 : \texttt{int}$
  Valeur de retour $x : \alpha$
  mais $\alpha = \texttt{int}$

- **let** $f = \texttt{fun}\ (x, y) \to\ x + y\ :\ \texttt{int} \times \texttt{int} \to \texttt{int}$
  $f : \alpha \times \beta \to \gamma$
  $(+) : \texttt{int} \to \texttt{int} \to \texttt{int}$
  d'où $\alpha = \texttt{int}$ et $\beta = \texttt{int}$ et $\gamma = \texttt{int}$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \rightarrow\ g\,(f\,x) : ?$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \rightarrow\ g\,(f\,x) : ?$
  $f : \alpha_1 \rightarrow \beta_1$
  $g : \alpha_2 \rightarrow \beta_2$
  $x : \alpha$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \rightarrow\ g\,(f\,x)$: ?

  $f : \alpha_1 \rightarrow \beta_1$

  $g : \alpha_2 \rightarrow \beta_2$

  $x : \cancel{\alpha} \alpha_1$

  $\alpha = \alpha_1$ d'où $\quad f\,x : \beta_1$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \rightarrow\ g\,(f\,x)$: ?

  $f : \alpha_1 \rightarrow \sout{\beta_1}\ \alpha_2$

  $g : \alpha_2 \rightarrow \beta_2$

  $x : \sout{\alpha}\ \alpha_1$

  $\alpha = \alpha_1$ d'où $\quad f\,x : \sout{\beta_1}\ \alpha_2$

  $\beta_1 = \alpha_2$ d'où $\quad g\,(f\,x) : \beta_2$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \rightarrow\ g\,(f\,x) : ?$

  $f : \alpha_1 \rightarrow \cancel{\beta_1}\ \alpha_2$

  $g : \alpha_2 \rightarrow \beta_2$

  $x : \cancel{\alpha}\ \alpha_1$

  $\alpha = \alpha_1$ d'où $\quad f\,x : \cancel{\beta_1}\ \alpha_2$

  $\beta_1 = \alpha_2$ d'où $\quad g\,(f\,x) : \beta_2$

  $\texttt{fun}\ x \rightarrow\ g\,(f\,x) : \alpha_1 \rightarrow \beta_2$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \to\ g\,(f\,x)$: ?

  $f : \alpha_1 \to \sout{\beta_1}\ \alpha_2$

  $g : \alpha_2 \to \beta_2$

  $x : \sout{\alpha}\ \alpha_1$

  $\alpha = \alpha_1$ d'où $\quad f\,x : \sout{\beta_1}\ \alpha_2$

  $\beta_1 = \alpha_2$ d'où $\quad g\,(f\,x) : \beta_2$

  $\texttt{fun}\ x \to\ g\,(f\,x) : \alpha_1 \to \beta_2$

  $\texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) : (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \to\ g\,(f\,x) : (\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$
  $f : \alpha_1 \to \beta_1\ \alpha_2$
  $g : \alpha_2 \to \beta_2$
  $x : \alpha\ \alpha_1$
  $\alpha = \alpha_1$ d'où $\quad f\,x : \beta_1\ \alpha_2$
  $\beta_1 = \alpha_2$ d'où $\quad g\,(f\,x) : \beta_2$
  $\texttt{fun}\ x \to\ g\,(f\,x) : \alpha_1 \to \beta_2$
  $\texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) : (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$
  $\texttt{fun}\ f \to\ \texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) :$
  $(\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \to\ g\,(f\,x) : (\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

  $f : \alpha_1 \to \beta_1\ \alpha_2$

  $g : \alpha_2 \to \beta_2$

  $x : \alpha\ \alpha_1$

  $\alpha = \alpha_1$ d'où $\quad f\,x : \beta_1\ \alpha_2$

  $\beta_1 = \alpha_2$ d'où $\quad g\,(f\,x) : \beta_2$

  $\texttt{fun}\ x \to\ g\,(f\,x) : \alpha_1 \to \beta_2$

  $\texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) : (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

  $\texttt{fun}\ f \to\ \texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) :$
  $(\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

- $\texttt{fun}\ x \to\ (\texttt{fun}\ g \to\ (\texttt{fun}\ f \to\ f\ x = g\ x))$

Inférence de types : exemples (2)

- $\texttt{fun}\ f\ g\ x \to\ g\,(f\,x) : (\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

  $f : \alpha_1 \to \beta_1\ \alpha_2$

  $g : \alpha_2 \to \beta_2$

  $x : \alpha\ \alpha_1$

  $\alpha = \alpha_1$ d'où $\quad f\,x : \beta_1\ \alpha_2$

  $\beta_1 = \alpha_2$ d'où $\quad g\,(f\,x) : \beta_2$

  $\texttt{fun}\ x \to\ g\,(f\,x) : \alpha_1 \to \beta_2$

  $\texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) : (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

  $\texttt{fun}\ f \to\ \texttt{fun}\ g \to\ \texttt{fun}\ x \to\ g\,(f\,x) :$
  $(\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

- $\texttt{fun}\ x \to\ (\texttt{fun}\ g \to\ (\texttt{fun}\ f \to\ f\ x = g\ x))$

  ```
  'a → ('a → 'b) → ('a → 'b) → bool
  ```

Exercices de typage

```
let flop = fun f x y -> f y x
```

Exercices de typage

```
let flop = fun f x y -> f y x

let rec compplus = fun l x -> match l with
| [] -> x
| f :: t -> f (compplus t x)
```

Exercices de typage

```
let flop = fun f x y -> f y x

let rec compplus = fun l x -> match l with
| [] -> x
| f :: t -> f (compplus t x)

let subst = fun f g x -> f x (g x)
```

Exercices de typage

```
let flop = fun f x y -> f y x

let rec compplus = fun l x -> match l with
| [] -> x
| f :: t -> f (compplus t x)

let subst = fun f g x -> f x (g x)

let machin = fun f g -> g (f g)
```

Plan

Plan

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste
- choisir un élément dans la liste
- choisir certains éléments dans la liste
- transformer chaque élément de la liste
- agréger les éléments de la liste

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste : α `list` $\rightarrow$ `bool`
- choisir un élément dans la liste
- choisir certains éléments dans la liste
- transformer chaque élément de la liste
- agréger les éléments de la liste

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste : α `list` $\rightarrow$ `bool`
- choisir un élément dans la liste : α `list` $\rightarrow \alpha$
- choisir certains éléments dans la liste
- transformer chaque élément de la liste
- agréger les éléments de la liste

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste : α `list` $\rightarrow$ `bool`
- choisir un élément dans la liste : α `list` $\rightarrow \alpha$
- choisir certains éléments dans la liste : α `list` $\rightarrow \alpha$ `list`
- transformer chaque élément de la liste
- agréger les éléments de la liste

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste : α `list` $\rightarrow$ `bool`
- choisir un élément dans la liste : α `list` $\rightarrow \alpha$
- choisir certains éléments dans la liste : α `list` $\rightarrow \alpha$ `list`
- transformer chaque élément de la liste : α `list` $\rightarrow \beta$ `list`
- agréger les éléments de la liste

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste : α `list` $\rightarrow$ `bool`
- choisir un élément dans la liste : α `list` $\rightarrow \alpha$
- choisir certains éléments dans la liste : α `list` $\rightarrow \alpha$ `list`
- transformer chaque élément de la liste : α `list` $\rightarrow \beta$ `list`
- agréger les éléments de la liste : α `list` $\rightarrow \beta$

Listes et ordre supérieur

De façon générale, que calcule une fonction à partir d'une liste ?
Le typage constitue un guide utile :

- tester une propriété sur toute la liste : α `list` $\rightarrow$ `bool`
- choisir un élément dans la liste : α `list` $\rightarrow \alpha$
- choisir certains éléments dans la liste : α `list` $\rightarrow \alpha$ `list`
- transformer chaque élément de la liste : α `list` $\rightarrow \beta$ `list`
- agréger les éléments de la liste : α `list` $\rightarrow \beta$

... pour ces fonctions.

Plan

Fonction quelquepart

Déetermine si au moins un élément d'une liste vérifie une propriété :

Fonction quelquepart

Déetermine si au moins un élément d'une liste vérifie une propriété :

```
let rec quelquepart = fun p l -> match l with
[ ] -> false
| x ::e -> (p x) || (quelquepart p e)
val quelquepart : ('a -> bool) -> 'a list -> bool = <fun>
# quelquepart (fun x -> x > 0) [1 ; 0 ; -1 ; 42] ;;
- : bool = true
```

Fonction quelquepart

Déetermine si au moins un élément d'une liste vérifie une propriété :

```
let rec quelquepart = fun p l -> match l with
[ ] -> false
| x ::e -> (p x) || (quelquepart p e)
val quelquepart : ('a -> bool) -> 'a list -> bool = <fun>
# quelquepart (fun x -> x > 0) [1 ; 0 ; -1 ; 42] ;;
- : bool = true
```

Prédéfinie : List.exists

Fonction partout

Détermine si tous les éléments d'une liste vérifient une propriété :

Fonction partout

Détermine si tous les éléments d'une liste vérifient une propriété :

```
let rec partout p l = match l with
[ ] → true
| x :: e → ( p x ) && (partout p e)
val partout : (α → bool) → α list → bool = <fun>
# partout (fun x → x > -2) [1 ; 0 ; -1 ; 42] ;;
- : bool = true
```

Fonction partout

Détermine si tous les éléments d'une liste vérifient une propriété :

```
let rec partout p l = match l with
[ ] → true
| x :: e → ( p x ) && (partout p e)
val partout : ('a → bool) → 'a list → bool = <fun>
# partout (fun x → x > -2) [1 ; 0 ; -1 ; 42] ;;
- : bool = true
```

Prédéfinie : List.forall

Fonction filter

Sélectionne dans une liste les éléments qui vérifient une propriété.

Fonction filter

Sélectionne dans une liste les éléments qui vérifient une propriété.

```
let rec filter = fun p l -> match l with
[ ] -> [ ]
| x :: e -> if p x then x :: filter p e else filter p e
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# filter (fun x -> x > 0) [1 ; 0 ; -1 ; 42] ;;
- : int list = [1 ; 42]
```

Fonction filter

Sélectionne dans une liste les éléments qui vérifient une propriété.

```
let rec filter = fun p l -> match l with
[] -> []
| x :: e -> if p x then x :: filter p e else filter p e
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# filter (fun x -> x > 0) [1; 0; -1; 42];;
- : int list = [1; 42]
```

Prédéfinie : List.filter

Fonction find

Sélectionne dans une liste le premier élément qui vérifie une propriété.

Fonction find

Sélectionne dans une liste le premier élément qui vérifie une propriété.

```
let rec find = fun p l -> match l with
[ ] -> ???
| x :: e -> if p x then x else find p e
val find : ('a -> bool) -> 'a list -> 'a = <fun>
```

Fonction find

Sélectionne dans une liste le premier élément qui vérifie une propriété.

```
let rec find = fun p l -> match l with
[ ] -> None
| x :: e -> if p x then Some x else find p e
val find : ('a -> bool) -> 'a list -> 'a option = <fun>
# find (fun x -> x < 0) [1 ; 0 ; -1 ; 42] ;;
- : int = Some (-1)
# find (fun x -> x > 100) [1 ; 0 ; -1 ; 42] ;;
- : int = None
```

Fonction find

Sélectionne dans une liste le premier élément qui vérifie une propriété.

```
let rec find = fun p l -> match l with
[ ] -> None
| x :: e -> if p x then Some x else find p e
val find : ('a -> bool) -> 'a list -> 'a option = <fun>
# find (fun x -> x < 0) [1 ; 0 ; -1 ; 42] ;;
- : int = Some (-1)
# find (fun x -> x > 100) [1 ; 0 ; -1 ; 42] ;;
- : int = None
```

Prédéfinie : List.find

Fonction map

Rend la liste obtenue en appliquant une même fonction à chaque élément de la liste en argument :

Fonction map

Rend la liste obtenue en appliquant une même fonction à chaque élément de la liste en argument :

```
let rec map = fun f l -> match l with
| [] -> []
| a :: l -> (f a) :: map f l

# map (fun x -> x + 1) [1; 0; -1; 42];;
- : int list = [2; 1; 0; 43]
```

Fonction map

Rend la liste obtenue en appliquant une même fonction à chaque élément de la liste en argument :

```
let rec map = fun f l -> match l with
| [] -> []
| a :: l -> (f a) :: map f l
```

```
# map (fun x -> x + 1) [1 ; 0 ; -1 ; 42] ;;
- : int list = [2 ; 1 ; 0 ; 43]
```

Prédéfinie : List.map

Fonction fold

Somme des entiers d'une liste :

Fonction fold

Somme des entiers d'une liste :

```
let rec somme = fun l -> match l with
| [ ] -> 0
| x :: l' -> x + somme l'
```

Produit des entiers d'une liste :

Fonction fold

Somme des entiers d'une liste :

```
let rec somme = fun l -> match l with
| [ ] -> 0
| x :: l' -> x + somme l'
```

Produit des entiers d'une liste :

```
let rec prod = fun l -> match l with
| [ ] -> 1
| x :: l' -> x * prod l'
```

Fonction fold (suite)

Longueur d'une liste :

Fonction fold (suite)

Longueur d'une liste :

```
let rec longueur = fun l -> match l with
| [ ] -> 0
| x :: l' -> 1 + longueur l'
```

Fonction fold (suite)

Longueur d'une liste :

```
let rec somme = fun l -> match l with
| [ ] -> 0
| x :: l' -> x + somme l'
```

Fonction fold (suite)

Longueur d'une liste :

```
let rec prod = fun l -> match l with
| [ ] -> 1
| x :: l' -> x * prod l'
```

Fonction fold (suite)

Longueur d'une liste :

```
let rec longueur = fun l -> match l with
| [ ] -> 0
| x :: l' -> 1 + longueur l'
```

Fonction fold (suite)

Longueur d'une liste :

```
let rec longueur = fun l -> match l with
| [ ] -> 0
| x :: l' -> 1 + longueur l'
```

Généralisation :

Fonction fold (suite)

Longueur d'une liste :

```
let rec longueur = fun l -> match l with
| [ ] -> 0
| x :: l' -> 1 + longueur l'
```

Généralisation :

```
let rec fold_right = fun op l e -> match l with
| [ ] -> e
| x :: l' -> op x (fold_right op l' e)
```

Fonction fold (suite)

Longueur d'une liste :

```
let rec longueur = fun l -> match l with
| [ ] -> 0
| x :: l' -> 1 + longueur l'
```

Généralisation :

```
let rec fold_right = fun op l e -> match l with
| [ ] -> e
| x :: l' -> op x (fold_right op l' e)
```

fold_right op [x1 ; x2 ; ... ; xn] e =
(op x1 (op x2 ... (op xn e)...))

Prédéfinie : List.fold_right

Une autre fonction fold

Dans fold_right :

- on commence les calculs « par la droite »

Une autre fonction fold

Dans fold_right :

- on commence les calculs « par la droite »

Autre possibilité : les commencer par la gauche

Une autre fonction fold

Dans fold_right :

- on commence les calculs « par la droite »

Autre possibilité : les commencer par la gauche

```
let rec somme = fun accu l -> match l with
| [ ] -> accu
| x :: l' -> somme (accu + x) l
in somme 0
```

Remarque : l'interpréteur OCaml détecte automatiquement la récursivité terminale pour en optimiser l'évaluation.

Fonction fold_left (1)

Produit des entiers d'une liste avec récursivité terminale :

Fonction fold_left (1)

Produit des entiers d'une liste avec récursivité terminale :

```
let rec prod = fun accu l -> match l with
| [ ] -> accu
| x :: l' -> prod (accu * x) l'
in prod 1
```

Calcul récursif terminal de la longueur d'une liste.

Fonction fold_left (1)

Produit des entiers d'une liste avec récursivité terminale :

```
let rec prod = fun accu l -> match l with
| [ ] -> accu
| x :: l' -> prod (accu * x) l'
in prod 1
```

Calcul récursif terminal de la longueur d'une liste.

Solution

```
let rec longueur accu l = match l with
| [ ] -> accu
| x :: l' -> longueur (accu + 1) l'
```

Fonction fold_left (2)

Généralisation :

Fonction fold_left (2)

Généralisation :

```
let rec fold_left = fun op accu l -> match l with
| [ ] -> accu
| x :: l' -> fold_left op (op accu x) l'
let somme = fold_left (fun x y -> x + y) 0 ;;
```

Fonction fold_left (2)

Généralisation :

```
let rec fold_left = fun op accu l -> match l with
| [ ] -> accu
| x :: l' -> fold_left op (op accu x) l'

let somme = fold_left (fun x y -> x + y) 0 ;;
let prod = fold_left (fun p x -> x * p) 1 ;;
let longueur = fold_left (fun l x -> l + 1) 0 ;;
```

Fonction fold_left (2)

Généralisation :

```
let rec fold_left = fun op accu l -> match l with
| [ ] -> accu
| x :: l' -> fold_left op (op accu x) l'

let somme = fold_left (fun x y -> x + y) 0 ;;
let prod = fold_left (fun p x -> x * p) 1 ;;
let longueur = fold_left (fun l x -> l + 1) 0 ;;
```

fold_left op a [x1 ; x2 ; ... ; xn] =
(op ... (op (op a x1) x2) ... xn)

Prédéfinie : List.fold_left