Programmation Fonctionnelle (PF)

INFO4

Cours 4 : typage et exceptions

Jean-François Monin, Benjamin Wack

POLYTECH GRENOBLE

2019 - 2020

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

Des exemples simples

fun x $\rightarrow$ x

Des exemples simples

fun x → x

$\text{int} \to \text{int}$

$\text{bool} \to \text{bool}$

$\alpha \to \alpha$

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

fun x $\rightarrow$ x x

Un exemple moins simple

fun x $\rightarrow$ x x

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

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 : succ $\qquad$ `int` $\rightarrow$ `int`

Types polymorphes (comparables à la généricité en Ada)

Exemple : test d'égalité $(=)$ $\qquad$ $\alpha \rightarrow \alpha \rightarrow$ `bool`

Typage dans un environnement

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

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

Exemples

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

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 *Pervasives*) 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.

Typage d'une expression

L'environnement Γ_0 initialement chargé (appelé *Pervasives*) 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 $\mathtt{s} : \mathtt{int} \to \mathtt{int}$
(et aucun autre couple $\mathtt{s} : t$)
vérifier que $\Gamma_0 \vdash \mathtt{fun\ n} \to \mathtt{s}\ (\mathtt{s}\ n) : \mathtt{int} \to \mathtt{int}$

Notation (pour raison de place)

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

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

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

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

Typage du let rec

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

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

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

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

Preuve et programme, même combat

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

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

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

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

Preuve et programme, même combat

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

$$\frac{\Gamma \vdash \quad t \qquad \Gamma \vdash \quad u}{\Gamma \vdash \qquad t \times u}$$

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

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

Preuve et programme, même combat

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

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

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

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

Preuve et programme, même combat

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

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

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

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

Preuve et programme, même combat

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

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

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

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

Preuve et programme, même combat

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

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

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

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

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

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

Preuve et programme, même combat

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

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

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

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

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

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

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

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

Preuve et programme, même combat

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

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

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

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

Preuve et programme, même combat

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

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

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

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

Système de typage (PF) Système de déduction (LT)

Que fait-on d'un système de types ?

- Vérification pure (type partout)
  fun (x :int) $\rightarrow$ (+ : int $\rightarrow$ int $\rightarrow$ int) (3 :int) (x :int) : int

Que fait-on d'un système de types ?

- Vérification pure (type 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 (type 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 (type 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 (type 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

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

Les mauvais élèves

- `fun x`$\to E$ demande de deviner le type de x
- mais la règle d'application peut aider
- **let** `rec` $x = A$ demande de deviner le type de x... et de A !

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 $\rightarrow$ x

type :

$\alpha \rightarrow \alpha$

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 $\rightarrow$ (n +3))
- à elle-même : id id, id id 3.14

Exemples basiques *

Fonction identité **let** id = `fun` x $\rightarrow$ x

type :

$\alpha \rightarrow \alpha$

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 $\rightarrow$ (n +3))
- à elle-même : id id, id id 3.14

Paire

let pairing = fun x y $\rightarrow$ (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

Paire

let pairing = fun x y → (x,y) ; ;

val id : '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 : $=$, $<$, $<=$, ...

$\alpha \to \alpha \to$ `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 $a\sigma = b\sigma$ (unificateur)
- le type inféré pour $f\ k$ est $c\sigma$

Unification de types Exemple

$\mathit{unifier}(\alpha \to \alpha, \mathit{int} \to \beta)$?

Unification de types Exemple

unifier$(\alpha \to \alpha, \mathit{int} \to \beta)$?

$\alpha = \mathit{int}$ 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 avec les mains

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

Inférence de types : exemples avec les mains

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

Inférence de types : exemples avec les mains

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

Inférence de types : exemples avec les mains

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

Inférence de types : exemples avec les mains

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

Inférence de types : exemples avec les mains

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

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

Inférence de types : exemples avec les mains

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

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

Inférence de types : exemples avec les mains

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

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

Inférence de types : exemples (2)

- `fun f g x` $\rightarrow$ $g\ (f\ x)$: ?

Inférence de types : exemples (2)

- `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)

- `fun f g x` $\rightarrow g\,(f\,x)$: ?

  $f : \alpha_1 \rightarrow \beta_1$

  $g : \alpha_2 \rightarrow \beta_2$

  $x : \alpha \leftarrow \alpha_1$

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

Inférence de types : exemples (2)

- `fun f g x` $\rightarrow g\,(f\,x)$: ?

  $f : \alpha_1 \rightarrow \beta_1\ \alpha_2$

  $g : \alpha_2 \rightarrow \beta_2$

  $x : \alpha\ \alpha_1$

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

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

Inférence de types : exemples (2)

- `fun f g x` $\rightarrow g\,(f\,x)$: ?

  $f : \alpha_1 \rightarrow \beta_1\ \alpha_2$

  $g : \alpha_2 \rightarrow \beta_2$

  $x : \alpha_1$

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

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

  `fun x` $\rightarrow g\,(f\,x) : \alpha_1 \rightarrow \beta_2$

Inférence de types : exemples (2)

- `fun f g x →` $g\ (f\ x)$: ?

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

  $g : \alpha_2 \to \beta_2$

  $x : \alpha\ \alpha_1$

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

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

  `fun x →` $g\ (f\ x) : \alpha_1 \to \beta_2$

  `fun g → fun x →` $g\ (f\ x) : (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

Inférence de types : exemples (2)

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

  $\alpha = \alpha_1$ d'où $\quad f\ x : \beta_1\ \alpha_2$
  $\beta_1 = \alpha_2$ d'où $\quad g\ (f\ x) : \beta_2$

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

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

Inférence de types : exemples (2)

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

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

Inférence de types : exemples (2)

- `fun f g x → g (f x)` : $(\alpha_1 \to \alpha_2) \to (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

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

  $g : \alpha_2 \to \beta_2$

  $x : \alpha_1$

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

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

  `fun x →` $g\ (f\ x) : \alpha_1 \to \beta_2$

  `fun g → fun x →` $g\ (f\ x) : (\alpha_2 \to \beta_2) \to \alpha_1 \to \beta_2$

- `fun x → (fun g → (fun f → f x = g x))`

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

Plan

Fonctions

Mathématiques : $x \mapsto f(x)$

OCaml : `fun x` $\rightarrow$ $f(x)$

Fonctions

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

Exemple : prédicat « est non nul »

Fonctions

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

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

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

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 "Curryfication"

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 arguments "Curryfication"

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

Exemples

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

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

`plus : int -> int -> int` qui se lit `int -> (int -> int)`

Fonctions à plusieurs arguments "Curryfication"

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

Exemples

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

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

`plus : int -> int -> int` qui se lit `int -> (int -> int)`

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

«Curryfication» exemple +

`int` × `int` → `int`

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

`int` → `int` → `int`

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

plusB : `int` → `int` → `int` qui se lit `int` → (`int` → `int`)

«Curryfication» exemple +

`int` × `int` → `int`

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

`int` → `int` → `int`

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

plusB : `int` → `int` → `int` qui se lit `int` → (`int` → `int`)

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 à 1 argument, par exemple

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

Quel est le résultat rendu par `aff23` en 1 ?

Les fonctions peuvent être en argument d'une fonction

On considère une fonction à 1 argument, par exemple

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

Quel est le résultat rendu par `aff23` en 1 ?

`aff23 1`

Les fonctions peuvent être en argument d'une fonction

On considère une fonction à 1 argument, par exemple
let `aff23` = `fun` x → 2*x + 3

Quel est le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire *f*

Quel est le résultat rendu par `f` en 1 ?

Les fonctions peuvent être en argument d'une fonction

On considère une fonction à 1 argument, par exemple
let `aff23` = `fun` x → 2*x + 3

Quel est le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire *f*

Quel est le résultat rendu par `f` en 1 ?

`f` 1

Les fonctions peuvent être en argument d'une fonction

On considère une fonction à 1 argument, par exemple
let `aff23` = `fun` x → 2*x + 3

Quel est le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire *f*

Quel est 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 à 1 argument, par exemple
let `aff23` = `fun` x → 2*x + 3

Quel est le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire *f*

Quel est 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 → *f* 1

Les fonctions peuvent être en argument d'une fonction

On considère une fonction à 1 argument, par exemple
let `aff23` = `fun` x → 2*x + 3

Quel est le résultat rendu par `aff23` en 1 ?

`aff23` 1

On se donne une fonction arbitraire *f*

Quel est 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 → *f* 1

Que rend `val1 aff23` ?

Application : terrain du robot jardinier

Ce qu'on veut

Associer à chaque case du terrain (coordonnées) un contenu

Application : terrain du robot jardinier

Ce qu'on veut

Associer à chaque case du terrain (coordonnées) un contenu

Solution classique

tableau à deux dimensions

- borné, erreurs sur les indices à gérer
- doit être initialisé
- problèmes de partage/copie
- limite le format des coordonnées

Application : terrain du robot jardinier

Ce qu'on veut

Associer à chaque case du terrain (coordonnées) un contenu

Solution classique

tableau à deux dimensions

- borné, erreurs sur les indices à gérer
- doit être initialisé
- problèmes de partage/copie
- limite le format des coordonnées

Solution fonctionnelle

type `etat_terrain` = `coordonnees` → `contenu_case`

- illimité
- initialisation triviale : **let** `terrain` = `fun c` → `Case_vide`
- paramétrable à volonté par les types `coordonnees` et `contenu_case`

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 des *citoyens de première classe*

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
- : int * int = (4, 5)
```

Modification du terrain par le robot jardinier

Ce qu'on veut

Étant donné un terrain et des coordonnées particulières :

- construire un nouveau terrain
- identique au précédent
- sauf pour ces coordonnées particulières

Modification du terrain par le robot jardinier

Ce qu'on veut

Étant donné un terrain et des coordonnées particulières :

- construire un nouveau terrain
- identique au précédent
- sauf pour ces coordonnées particulières

```
let deposer = fun (obj : objet) →
    fun (coord : coordonnees) →
    fun (terrain : etat_terrain) →
```

Modification du terrain par le robot jardinier

Ce qu'on veut

Étant donné un terrain et des coordonnées particulières :

- construire un nouveau terrain
- identique au précédent
- sauf pour ces coordonnées particulières

```
let deposer = fun (obj : objet) →
    fun (coord : coordonnees) →
    fun (terrain : etat_terrain) →
        fun c → ...
```

Modification du terrain par le robot jardinier

Ce qu'on veut

Étant donné un terrain et des coordonnées particulières :

- construire un nouveau terrain
- identique au précédent
- sauf pour ces coordonnées particulières

```
let deposer = fun (obj : objet) →
    fun (coord : coordonnees) →
    fun (terrain : etat_terrain) →
        fun c →
                terrain c
```

Modification du terrain par le robot jardinier

Ce qu'on veut

Étant donné un terrain et des coordonnées particulières :

- construire un nouveau terrain
- identique au précédent
- sauf pour ces coordonnées particulières

```
let deposer = fun (obj : objet) →
    fun (coord : coordonnees) →
    fun (terrain : etat_terrain) →
        fun c →  if c = coord then Case_obj(obj)
                 else terrain c
```

Conclusion

Aujourd'hui

- Typage
- Polymorphisme
- Inférence de type
- Ordre supérieur

La prochaine fois

- Grammaires et analyse descendante naïve
- Flots
- Analyse lexicale et analyse syntaxique

Juste pour chercher les ennuis...

```
let flop = fun f x y → f y x
```

Juste pour chercher les ennuis...

```
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)
```

Juste pour chercher les ennuis...

```
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)
```

Juste pour chercher les ennuis...

```
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)
```

Récursivité, ordre supérieur et typage

Quel est le type de cette fonction ? Que calcule-t-elle ?

```
let rec power = fun f n x -> if n = 0 then x
                             else f (power f (n-1) x)
```

Récursivité, ordre supérieur et typage

Quel est le type de cette fonction ? Que calcule-t-elle ?

```
let rec power = fun f n x -> if n = 0 then x
                             else f (power f (n-1) x)
```

```
- : ('a → 'a) → int → 'a → 'a = <fun>
```

Récursivité, ordre supérieur et typage

Quel est le type de cette fonction ? Que calcule-t-elle ?

```
let rec power = fun f n x -> if n = 0 then x
                             else f (power f (n-1) x)
```

```
- : ('a → 'a) → int → 'a → 'a = <fun>
```

Version récursive terminale ?

Récursivité, ordre supérieur et typage

Quel est le type de cette fonction ? Que calcule-t-elle ?

```
let rec power = fun f n x -> if n = 0 then x
                             else f (power f (n-1) x)
```

```
- : ('a → 'a) → int → 'a → 'a = <fun>
```

Version récursive terminale ?

```
let rec power = fun f n x → if n = 0 then x
                            else power f (n-1) (f x)
```

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

- α `list` $\rightarrow$ `bool`
- α `list` $\rightarrow \alpha$
- α `list` $\rightarrow \alpha$ `list`
- α `list` $\rightarrow \beta$ `list`
- α `list` $\rightarrow \beta$

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

- α `list` $\rightarrow$ `bool` : tester une propriété sur toute la liste
- α `list` $\rightarrow \alpha$
- α `list` $\rightarrow \alpha$ `list`
- α `list` $\rightarrow \beta$ `list`
- α `list` $\rightarrow \beta$

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

- α `list` $\rightarrow$ `bool` : tester une propriété sur toute la liste
- α `list` $\rightarrow \alpha$: choisir un élément dans la liste (! partielle !)
- α `list` $\rightarrow \alpha$ `list`
- α `list` $\rightarrow \beta$ `list`
- α `list` $\rightarrow \beta$

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

- α `list` $\rightarrow$ `bool` : tester une propriété sur toute la liste
- α `list` $\rightarrow \alpha$: choisir un élément dans la liste (! partielle !)
- α `list` $\rightarrow \alpha$ `list` : choisir certains éléments dans la liste
- α `list` $\rightarrow \beta$ `list`
- α `list` $\rightarrow \beta$

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

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

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

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

Listes et ordre supérieur

De façon générale, que peut faire une fonction avec une liste ?
Le typage apporte un début de réponse :

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

... exprimé par une fonction.

exists

Écrire une fonction qui détermine si au moins un élément d'une liste vérifie une propriété :

exists

Écrire une fonction qui détermine si au moins un élément d'une liste vérifie une propriété :

```
let rec exists = fun p l -> match l with
[ ] -> false
| x : :e -> (p x) || (exists p e)
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

```
# exists (fun x -> x > 0) [1 ;0 ;-1 ;42] ; ;
- : bool = true
```

exists

Écrire une fonction qui détermine si au moins un élément d'une liste vérifie une propriété :

```
let rec exists = fun p l -> match l with
[ ] -> false
| x : :e -> (p x) || (exists p e)
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

```
# exists (fun x -> x > 0) [1 ;0 ;-1 ;42] ; ;
- : bool = true
```

Prédéfinie : List.exists

voir aussi List.forall

filter

Écrire une fonction qui sélectionne dans une liste les éléments qui vérifient une propriété.

filter

Écrire une fonction qui 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 : (α -> bool) -> α list -> α list = <fun>
```

```
# filter (fun x -> x > 0) [1 ;0 ;-1 ;42] ; ;
- : int list = [1 ; 42]
```

filter

Écrire une fonction qui 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 : (α -> bool) -> α list -> α list = <fun>
```

```
# filter (fun x -> x > 0) [1 ;0 ;-1 ;42] ; ;
- : int list = [1 ; 42]
```

Prédéfinie : List.filter

find

Écrire une fonction qui sélectionne dans une liste le premier élément qui vérifie une propriété.

find

Écrire une fonction qui 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 : (α -> bool) -> α list -> α  = <fun>
```

find

Écrire une fonction qui 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 : (α -> bool) -> α list -> α 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
```

find

Écrire une fonction qui 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 : (α → bool) → α list → α 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

map

Écrire une fonction qui transforme une liste en une autre en appliquant une même fonction à chaque élément :

map

Écrire une fonction qui transforme une liste en une autre en appliquant une même fonction à chaque élément :

```
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]
```

map

Écrire une fonction qui transforme une liste en une autre en appliquant une même fonction à chaque élément :

```
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

fold

Écrire une fonction qui calcule la somme des entiers d'une liste :

fold

Écrire une fonction qui calcule la somme des entiers d'une liste :

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

Écrire une fonction qui calcule le produit des entiers d'une liste :

fold

Écrire une fonction qui calcule la somme des entiers d'une liste :

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

Écrire une fonction qui calcule le produit des entiers d'une liste :

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

fold (suite)

Écrire une fonction qui calcule la longueur d'une liste :

fold (suite)

Écrire une fonction qui calcule la longueur d'une liste :

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

fold (suite)

Écrire une fonction qui calcule la longueur d'une liste :

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

fold (suite)

Écrire une fonction qui calcule la longueur d'une liste :

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

fold (suite)

Écrire une fonction qui calcule la longueur d'une liste :

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

fold (suite)

Écrire une fonction qui calcule la longueur d'une liste :

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

Généralisation :

fold (suite)

Écrire une fonction qui calcule la 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 e -> match l with
| [] -> e
| x::l' -> op x (fold_right op l' e)
```

fold (suite)

Écrire une fonction qui calcule la 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`

Un autre fold

Problème de `fold_right` : intuitif mais

- e ne « sert à rien » avant la fin de la liste
- on commence les calculs « par la droite »

Un autre fold

Problème de `fold_right` : intuitif mais

- e ne « sert à rien » avant la fin de la liste
- on commence les calculs « par la droite »

Proposition : récursivité terminale

Un autre fold

Problème de `fold_right` : intuitif mais

- *e* ne « sert à rien » avant la fin de la liste
- on commence les calculs « par la droite »

Proposition : récursivité terminale

```
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 cette forme pour en optimiser l'évaluation.

fold_left

Écrire une fonction qui calcule le produit des entiers d'une liste :

fold_left

Écrire une fonction qui calcule le produit des entiers d'une liste :

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

Généraliser :

fold_left

Écrire une fonction qui calcule le produit des entiers d'une liste :

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

Généraliser :

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

fold_left

Écrire une fonction qui calcule le produit des entiers d'une liste :

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

Généraliser :

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

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

fold_left

Écrire une fonction qui calcule le produit des entiers d'une liste :

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

Généraliser :

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

```
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`