Introduction à Objective Caml

Emmanuel Filiot

25 novembre 2004

Plan

- famille ML (SML, Caml, Caml-light)
- ancêtre : Caml (1986-1990), projet Formel Inria
- puis OCaml, projet Cristal

- famille ML (SML, Caml, Caml-light)
- ancêtre : Caml (1986-1990), projet Formel Inria
- puis OCaml, projet Cristal

- famille ML (SML, Caml, Caml-light)
- ancêtre : Caml (1986-1990), projet Formel Inria
- puis OCaml, projet Cristal

- `http ://caml.inria.fr`
- paquets Debian, Mandrake ...

- `http ://caml.inria.fr`
- paquets Debian, Mandrake ...

Le toplevel

- boucle d'interprétation (*toplevel*)

```
efiliot@platpays % ocaml
# let a = 1 in a + 2 ;;
- : int = 3
```

- ou mieux :

```
efiliot@platpays % ledit ocaml
```

- encore mieux : **tuareg-mode**, un mode Caml pour Emacs
  `C-c C-e` pour interpréter
  TODO : référence à donner démo

Dans ce cours on utilisera le toplevel

Compilation

À partir de fichier .ml pour le code, .mli pour les interfaces.

- bytecode : `ocamlc`, produit rapidement du code interprété
- code natif : `ocamlopt`, produit du code machine, plus efficace
- voir le OCamlMakefile TODO : référence à donner

Compilation

À partir de fichier .ml pour le code, .mli pour les interfaces.

- bytecode : `ocamlc`, produit rapidement du code interprété
- code natif : `ocamlopt`, produit du code machine, plus efficace
- voir le OCamlMakefile TODO : référence à donner

Compilation

À partir de fichier .ml pour le code, .mli pour les interfaces.

- bytecode : `ocamlc`, produit rapidement du code interprété
- code natif : `ocamlopt`, produit du code machine, plus efficace
- voir le OCamlMakefile TODO : référence à donner

Des références

- le manuel de référence sur `http ://caml.inria.fr`
- le bouquin O'Reilly *Développement d'application avec Objective Caml*
- les slides d'un cours de programmation sur `http ://perso.ens-lyon.fr/daniel.hirschkoff`, avec des références

Des références

- le manuel de référence sur `http ://caml.inria.fr`
- le bouquin O'Reilly *Développement d'application avec Objective Caml*
- les slides d'un cours de programmation sur `http ://perso.ens-lyon.fr/daniel.hirschkoff`, avec des références

Des références

- le manuel de référence sur `http ://caml.inria.fr`
- le bouquin O'Reilly *Développement d'application avec Objective Caml*
- les slides d'un cours de programmation sur `http ://perso.ens-lyon.fr/daniel.hirschkoff`, avec des références

Description du langage

- **langage fonctionnel** : les fonctions sont des objets de première classe, au même titre que les entiers, les chaînes, etc...
- **statiquement typé** La vérification des types est faite à la compilation. Nul besoin de les indiquer dans le code, le moteur d'inférence de type s'en charge. Plus d'erreurs à la compilation, moins à l'exécution. Les types donnent une sémantique des expressions, utile pour les bibliothèques.
- **polymorphisme** OCaml accepte des expressions dont le type n'est pas entièrement déterminé :
  `type 'a list`
  $\longrightarrow$ *réutilisabilité du code*

Description du langage

- **langage fonctionnel** : les fonctions sont des objets de première classe, au même titre que les entiers, les chaînes, etc...
- **statiquement typé** La vérification des types est faite à la compilation. Nul besoin de les indiquer dans le code, le moteur d'inférence de type s'en charge. Plus d'erreurs à la compilation, moins à l'exécution. Les types donnent une sémantique des expressions, utile pour les bibliothèques.
- **polymorphisme** OCaml accepte des expressions dont le type n'est pas entièrement déterminé :
  `type 'a list`
  $\longrightarrow$ *réutilisabilité du code*

Description du langage

- **langage fonctionnel** : les fonctions sont des objets de première classe, au même titre que les entiers, les chaînes, etc...
- **statiquement typé** La vérification des types est faite à la compilation. Nul besoin de les indiquer dans le code, le moteur d'inférence de type s'en charge. Plus d'erreurs à la compilation, moins à l'exécution. Les types donnent une sémantique des expressions, utile pour les bibliothèques.
- **polymorphisme** OCaml accepte des expressions dont le type n'est pas entièrement déterminé :
  `type 'a list`
  $\longrightarrow$ *réutilisabilité du code*

Description du langage

- **mécanisme d'exception** : rompre l'exécution normale d'un programme
- **aspect impératifs** : entrées-sorties, structure de contrôle, références ...
- **programmation modulaire** fragments de code compilables séparément
- **programmation objet**
- threads, flots, lexer, parser, debugger, profiler, ...

Description du langage

- **mécanisme d'exception** : rompre l'exécution normale d'un programme
- **aspect impératifs** : entrées-sorties, structure de contrôle, références ...
- **programmation modulaire** fragments de code compilables séparément
- **programmation objet**
- threads, flots, lexer, parser, debugger, profiler, ...

Description du langage

- **mécanisme d'exception** : rompre l'exécution normale d'un programme
- **aspect impératifs** : entrées-sorties, structure de contrôle, références ...
- **programmation modulaire** fragments de code compilables séparément
- **programmation objet**
- threads, flots, lexer, parser, debugger, profiler, ...

Description du langage

- **mécanisme d'exception** : rompre l'exécution normale d'un programme
- **aspect impératifs** : entrées-sorties, structure de contrôle, références ...
- **programmation modulaire** fragments de code compilables séparément
- **programmation objet**
- threads, flots, lexer, parser, debugger, profiler, ...

Description du langage

- **mécanisme d'exception** : rompre l'exécution normale d'un programme
- **aspect impératifs** : entrées-sorties, structure de contrôle, références ...
- **programmation modulaire** fragments de code compilables séparément
- **programmation objet**
- threads, flots, lexer, parser, debugger, profiler, ...

Plan

λ-calcul

Modèle de programmation fonctionnelle : le λ-calcul.

$$\begin{array}{lll} M ::= & & \lambda\text{-terme} \\ & x & \text{variable} \\ & \mid MN & \text{application} \\ & \mid \lambda x.M & \text{abstraction} \end{array}$$

Une relation pour le calcul : la β-réduction notée $\rightarrow_\beta$.

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

- Turing-complet
- λ-terme = programme fonctionnel

λ-calcul

Modèle de programmation fonctionnelle : le λ-calcul.

$$\begin{array}{lll} M ::= & & \lambda\text{-terme} \\ & x & \text{variable} \\ & \mid MN & \text{application} \\ & \mid \lambda x.M & \text{abstraction} \end{array}$$

Une relation pour le calcul : la β-réduction notée $\rightarrow_\beta$.

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

- Turing-complet
- λ-terme $=$ programme fonctionnel

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta$
- $((\lambda x.\lambda y.x)(\lambda x.x))M \rightarrow_\beta (\lambda y.(\lambda x.x))M \rightarrow_\beta (\lambda x.x)$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \ \rightarrow_\beta$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta y$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta y$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé : si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul

Les fonctions sont des citoyens de première classe : elle peuvent être prises en argument, et être la valeur retournée par une autre fonction.

- $\lambda x.M$ peut être vu comme la fonction $x \mapsto M$
- et $\lambda x.\lambda y.x$?
- $(\lambda x.x)y \rightarrow_\beta y$
- $$\begin{array}{ll} ((\lambda x.\lambda y.x)(\lambda x.x))M & \rightarrow_\beta \\ (\lambda y.(\lambda x.x))M & \rightarrow_\beta \\ (\lambda x.x) & \end{array}$$
- et $(\lambda x.xx)(\lambda x.xx)$?
  $\rightarrow$ restriction au λ-calcul simplement typé :
  si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \rightarrow \tau'$

λ-calcul et OCaml

si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \longrightarrow \tau'$

Quel est le type de $\lambda x.\lambda y.x$ sachant que $x : \tau$ et $y : \tau'$?

Donner un λ-terme dont le type est
$(p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r$.

λ-calcul et OCaml

si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \to \tau'$

Quel est le type de $\lambda x.\lambda y.x$ sachant que $x : \tau$ et $y : \tau'$?

$\tau \to (\tau' \to \tau)$ noté $\tau \to \tau' \to \tau$

Donner un λ-terme dont le type est
$(p \to q \to r) \to (p \to q) \to p \to r$.

λ-calcul et OCaml

si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \to \tau'$

Quel est le type de $\lambda x.\lambda y.x$ sachant que $x : \tau$ et $y : \tau'$?

$\tau \to (\tau' \to \tau)$ noté $\tau \to \tau' \to \tau$

Donner un λ-terme dont le type est
$(p \to q \to r) \to (p \to q) \to p \to r$.

$\lambda f.\lambda g.\lambda x.fx(gx)$ avec $x : p$, $g : p \to q$ et $f : p \to q \to r$

λ-calcul et OCaml

si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \to \tau'$

Quel est le type de $\lambda x.\lambda y.x$ sachant que $x : \tau$ et $y : \tau'$?

$\tau \to (\tau' \to \tau)$ noté $\tau \to \tau' \to \tau$

Donner un λ-terme dont le type est
$(p \to q \to r) \to (p \to q) \to p \to r$.

$\lambda f.\lambda g.\lambda x.fx(gx)$ avec $x : p$, $g : p \to q$ et $f : p \to q \to r$

en OCaml : fun x → fun y → x : 'a → 'b → 'a

démo

λ-calcul et OCaml

si $x : \tau$ et $M : \tau'$, alors $\lambda x.M : \tau \to \tau'$

Quel est le type de $\lambda x.\lambda y.x$ sachant que $x : \tau$ et $y : \tau'$?

$\tau \to (\tau' \to \tau)$ noté $\tau \to \tau' \to \tau$

Donner un λ-terme dont le type est
$(p \to q \to r) \to (p \to q) \to p \to r$.

$\lambda f.\lambda g.\lambda x.fx(gx)$ avec $x : p$, $g : p \to q$ et $f : p \to q \to r$

en OCaml : fun x → fun y → x : 'a → 'b → 'a

démo

λ-calcul : trop limité → ajout de valeurs de base (entiers,...),structures de contrôle, affectation, nommage des expressions, de la récursivité

λ-calcul : réduction du sujet et forte normalisation

Théorème[Subject Reduction] Si $t \rightarrow_\beta t'$ et $t : \tau$, alors $t' : \tau$.

Ce théorème dit que le typage statique, c'est pas n'importe quoi. En plus, OCaml devine les types.

Théorème Tout terme t typable est fortement normalisable, i.e. qu'il n'existe pas de suite infinie de réductions $t \rightarrow_\beta t_1 \rightarrow_\beta ... \rightarrow_\beta t_n \rightarrow_\beta ...$, i.e. "l'évaluation de tout terme typable termine".

Noyau fonctionnel

- Un `programme` est une `expression`
- `l'exécution` d'un programme est l'évaluation de l'expression correspondante.
- En λ-calcul, expression $=$ λ-terme, et évalution $=$ application répétée de la β-réduction
- En OCaml, on n'écrit pas une suite d'instrution,mais on nomme des expressions avec `let`

Noyau fonctionnel

- Un `programme` est une `expression`
- `l'exécution` d'un programme est l'évaluation de l'expression correspondante.
- En λ-calcul, expression $=$ λ-terme, et évalution $=$ application répétée de la β-réduction
- En OCaml, on n'écrit pas une suite d'instrution,mais on nomme des expressions avec `let`

Noyau fonctionnel

- Un `programme` est une `expression`
- `l'exécution` d'un programme est l'évaluation de l'expression correspondante.
- En λ-calcul, expression $=$ λ-terme, et évalution $=$ application répétée de la β-réduction
- En OCaml, on n'écrit pas une suite d'instrution,mais on nomme des expressions avec `let`

Noyau fonctionnel

- Un `programme` est une `expression`
- `l'exécution` d'un programme est l'évaluation de l'expression correspondante.
- En λ-calcul, expression $=$ λ-terme, et évalution $=$ application répétée de la β-réduction
- En OCaml, on n'écrit pas une suite d'instrution,mais on nomme des expressions avec `let`

Premier exemple : la fonction factorielle


```
let rec fact n =
    if n = 0 then 1
    else n * fact (n-1)
```

Premier exemple : la fonction factorielle avec pattern-matching


```
let rec fact2 n = match n with
  0 -> 1
| _ -> n * fact2 (n-1)
```

Premier exemple : la fonction factorielle troisième version


```
let fact3 n =
    let rec aux acc m = match m with
        0 -> acc
      | p -> aux (p * acc) (p-1)
    in
    aux 1 n
```

Évaluation


```
# "toto";;
- : string = "toto"
```

À gauche, le type, à droite, la valeur.

```
# fun x -> x + 1 ;;
- : int -> int = <fun>
```

Valeur d'une séquence, valeur d'un effet de bord

`e1 ;e2 ;... ;en`

La valeur d'une séquence est la valeur de sa dernière expression

$\longrightarrow$ démo.

La valeur d'un effet de bord est `()`, de type `unit` (qui ne contient que cette valeur). Il "correspond" un peu au type void.

$\longrightarrow$ simulation des procédures

```
# print_string "coin\n" ;;
coin
- : unit = ()
```

Variable locale

`let nom = e1 in e2`

L'évaluation de e2 se fait dans un environnement dans lequel `nom` est déclaré. Comment évaluer une telle expression ?

- première stratégie : évaluer e1 puis e2 en remplaçant `nom` par la valeur de e1
- deuxième stratégie : remplacer `nom` par e1 dans e2 puis évaluer e2
- en OCaml, première stratégie (démo)
- la portée de `nom` est limitée à e2 (*nom* n'est connu que pour l'évaluation de e2)

Variable locale

`let nom = e1 in e2`

L'évaluation de e2 se fait dans un environnement dans lequel `nom` est déclaré. Comment évaluer une telle expression ?

- première stratégie : évaluer `e1` puis `e2` en remplaçant `nom` par la valeur de `e1`
- deuxième stratégie : remplacer `nom` par `e1` dans `e2` puis évaluer `e2`
- en OCaml, première stratégie (démo)
- la portée de `nom` est limitée à `e2` (*nom* n'est connu que pour l'évaluation de `e2`)

Variable locale

`let nom = e1 in e2`

L'évaluation de e2 se fait dans un environnement dans lequel `nom` est déclaré. Comment évaluer une telle expression ?

- première stratégie : évaluer `e1` puis `e2` en remplaçant `nom` par la valeur de `e1`
- deuxième stratégie : remplacer `nom` par `e1` dans `e2` puis évaluer `e2`
- en OCaml, première stratégie (démo)
- la portée de `nom` est limitée à e2 (*nom* n'est connu que pour l'évaluation de e2)

Variable locale

`let nom = e1 in e2`

L'évaluation de e2 se fait dans un environnement dans lequel `nom` est déclaré. Comment évaluer une telle expression ?

- première stratégie : évaluer `e1` puis `e2` en remplaçant `nom` par la valeur de `e1`
- deuxième stratégie : remplacer `nom` par `e1` dans `e2` puis évaluer `e2`
- en OCaml, première stratégie (démo)
- la portée de `nom` est limitée à `e2` (*nom* n'est connu que pour l'évaluation de `e2`)

Variable locale

`let nom = e1 in e2`

L'évaluation de e2 se fait dans un environnement dans lequel `nom` est déclaré. Comment évaluer une telle expression ?

- première stratégie : évaluer `e1` puis `e2` en remplaçant `nom` par la valeur de `e1`
- deuxième stratégie : remplacer `nom` par `e1` dans `e2` puis évaluer `e2`
- en OCaml, première stratégie (démo)
- la portée de `nom` est limitée à `e2` (*nom* n'est connu que pour l'évaluation de `e2`)

Valeurs, type de base


```
# 1 ;; (* entiers *)
- : int = 1
# "coucou" ;; (* chaînes *)
- : string = "coucou"
# 1. ;; (* flottants *)
- : float = 1.
# true ;; (* booléens *)
- : bool = true
# () ;; (* unité *)
- : unit = ()
```

Produit Cartésien


```
# (7,"fois") ;;
- : int * string = (7, "fois")
# let a = 1 in (a,a),a ;;
- : (int * int) * int = ((1, 1), 1)
# let b = "true" in (b,b,b) ;;
- : string * string * string = ("true", "true", "true
# fst ("premier", "second") ;;
- : string = "premier"
# snd ("premier", "second") ;;
- : string = "second"
```

Les listes


```
# [] ;; (* liste vide *)
- : 'a list = []
# [1;3;4] ;; (* liste d'entiers *)
- : int list = [1; 3; 4]
# let a = [ 1 ; 2 ] in 30 :: a ;;
(* ajout d'un élément en tête *)
- : int list = [30; 1; 2]
# let b = [ -1 ; -4 ] in b @ a ;; (* concaténation *)
- : int list = [-1; -4; 30; 1; 2]
# [ 1 ; true ] ;;
true : This expression has type bool
but is here used with type int (* démo ici *)
```

$\longrightarrow$ éléments d'un même type

Structure de contrôle

`if` *cond* `then` e_1 `else` e_2

```
# if true then 1 else 2 ;;
- : int = 1
# if false then "impossible" else 3 ;;
3 : This expression has type int but is here used
    with type string
# if false then print_string "pas ici" ;;
- : unit = ()
```

$\longrightarrow$ $\text{type}(e_1) = \text{type}(e_2)$

$\longrightarrow$ omission de la branche `else` : elle est remplacée par `else ()`

Les fonctions

`let f x =` e_1 ou `let f = fun x ->` e_1 ou `let f = function x ->` e_1

Si la fonction est récursive, on ajoute le mot clé rec (`let rec f x =` e_1)

Le type d'une fonction $x \mapsto e_1$ est $type(x) \longrightarrow type(e_1)$:

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

L'application d'une fonction s'écrit simplement `f x`

Fonctions à plusieurs arguments

Deux solutions :

- prendre un tuple en argument (forme *décurryfiée*)

```
# let f (x,y) = x + y ;;
val f : int * int -> int = <fun>
```

- écrire une fonction qui retourne une fonction qui retourne une fonction qui ... (forme *curryfiée*)

```
# let f x  = fun y -> x + y ;;
val f : int -> int -> int = <fun>
```

qui se note plus simplement `let f x y = x + y`

Fonctions mutuellement récursives


```
let rec f x =
  if x = 0 then true
  else if x = 1 then false
       else g (x-1)
and g x =
  if x = 0 then false
  else if x = 1 then true
       else f (x-1)
```

Que renvoient `f` et `g` ?

Voir aussi la démo sur *l'application partielle*

Fonctions infixes

(op)

```
# let (%) x y = x - y*y ;;
val ( % ) : int -> int -> int = <fun>
# 1 % 2 ;;
- : int = -3
# let (+) x y = x ;;
val ( + ) : 'a -> 'b -> 'a = <fun>
# [] + 2 ;;
- : 'a list = []
```

Exercices

- deviner le type de `let d x y z = if x (y+1) then z 5`
- écrire une fonction qui prend une fonction sous forme décurryfiée en argument, et qui retourne la même fonction sous forme curryfiée

Polymorphisme

Exemple, on veut écrire la fonction identité, a-t-on besoin de connaître le type de l'argument ?

Polymorphisme

Exemple, on veut écrire la fonction identité, a-t-on besoin de connaître le type de l'argument ? `let id x = x ; ;`

```
val id : 'a -> 'a = <fun>
```

signifie *"pour un élément de type quelconque `'a`, la fonction `id` retourne un élément du même type"*.

Polymorphisme

Exemple, on veut écrire la fonction identité, a-t-on besoin de connaître le type de l'argument ? `let id x = x ; ;`

```
val id : 'a -> 'a = <fun>
```

signifie *"pour un élément de type quelconque `'a`, la fonction `id` retourne un élément du même type"*.

On utilise la même fonction pour des arguments de types différents $\longrightarrow$ réutilisabilité du code

Polymorphisme (2)


```
 let couple x y = (x,y) ; ;
val couple : 'a -> 'b -> 'a*'b = <fun>
```

(démo)

$\longrightarrow$ le moteur d'inférence de type choisit le type le plus général : en parcourant les expressions, il génère des contraintes de type (équations) et en fait la résolution (unification).

Exercice Classer les types suivant du moins général au plus général : `int*'a`, `int*bool`, et `'a*'b`.

Polymorphisme (2)


```
 let couple x y = (x,y) ; ;
val couple : 'a -> 'b -> 'a*'b = <fun>
```

(démo)

Par contre :

```
let couple_eq x y = if x=y then (x,y) else (y,x) ; ;
val couple_eq : 'a -> 'a -> 'a*'a = <fun>
```

$\longrightarrow$ le moteur d'inférence de type choisit le type le plus général : en parcourant les expressions, il génère des contraintes de type (équations) et en fait la résolution (unification).

Exercice Classer les types suivant du moins général au plus général : `int*'a`, `int*bool`, et `'a*'b`.

Contrainte de type


```
let id_int (x : int) = x ;;
```

- voir le type
- contraindre l'utilisation de la fonction (`id_int true` pas possible)

Déclaration de types


```
type nom_type = un_type
```

nom_type est une abréviation pour un_type.

exemple :

```
# type couple_entiers = int * int ;;
type couple_entiers = int * int
# let f ( x : couple_entiers) = fst x ;;
val f : couple_entiers -> int = <fun>
```

démo

types paramétrés

`type ('a1,...,'an) nom_type = un_type`

`'a`$_i$: variables de type

```
# type 'a t = 'a * 'a ;;
type 'a t = 'a * 'a
```

démo

Enregistrements

n-uplets avec champs nommés.

```
# type toto = { titi : int ; tata : bool } ;;
type toto = { titi : int; tata : bool; }
# let x = { titi = 3 ; tata = true } ;;
val x : toto = {titi = 3; tata = true}
# x.titi ;;
- : int = 3
# x.tata ;;
- : bool = true
```

$\longrightarrow$ on ne peut pas modifier la valeur des champs

Types somme

- regrouper dans un même type des types différents, à l'aide de *constructeurs*
- les constructeurs permettent d'accéder aux valeurs de ce type, via le *pattern-matching*
- les constructeurs permettent de construire les valeurs du type
- les noms de constructeurs commencent toujours par une majuscule

```
type sexe = Masculin | Feminin
type age = EnGestation | PasEnGestation of int
type 'a un_ou_deux = Un of 'a | Deux of 'a * 'a
type 'a option = Some of 'a | None
type nat = Zero | Succ of nat
```

Types somme (2)


```
# type 'a t = A | B of 'a ;;
type 'a t = A | B of 'a
# let a = A ;;
val a : 'a t = A
# let b = B [] ;;
val b : 'a list t = B []
```

Types somme (3)

Pattern-matching : accéder aux valeurs du type

```
# let f x = match x with A -> [ 1;2;3] | B k -> List.
val f : int list t -> int list = <fun>
# f A ;;
- : int list = [1; 2; 3]
# f (B (f A)) ;;
- : int list = [3; 2; 1]
```

Types somme (4)


```
# type t = A | B ;;
type t = A | B
# type tt = A | C ;;
type tt = A | C
# let f x = match x with
    A -> 1
  | B -> 2 ;;
This pattern matches values of type t
but is here used to match values of type tt
```

Types somme (5)


```
# type t = A | B ;;
type t = A | B
# let f x = match x with A -> 1 ;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
B
val f : t -> int = <fun>
```

Pattern-matching

Déstructurer les valeurs à l'aide de motifs (*patterns*).
Le motif universel est _

`match` *expr* `with`
motif$_1$ `->` *expr*$_1$
motif$_2$ `->` *expr*$_2$
...
motif$_n$ `->` *expr*$_n$

L'expression *expr* est confrontée aux motifs (matching), dans l'ordre, si elle satisfait le motif, alors l'expression correspondante est évaluée.
Tout expression satisfait le motif _

Pattern-matching : des exemples

voir démo

Les exceptions

$\longrightarrow$ *Rompre le flot d'exécution du programme.*

$\longrightarrow$ *Comme méthode de calcul*

```
exception nom_constructeur
exception nom_constructeur of nom_type
```

- **Lever une exception** `raise exception`
- **Récupérer une exception** `try expr with exception1 -> ... | exception2 -> ...`

démo

Récursivité terminale

OCaml gère la récursivité terminale (*tail-recursion*)

La dernière chose qui est faite est le premier appel récursif

Un exemple (pas de récursivité terminale) :

```
let rec fact = function
  0 -> 1
| n -> n * (fact (n-1))
(* la dernière chose qui est faite *)
(* ici est la multiplication *)
```

Pour calculer `fact`, les appels à `fact` s'empilent. Si `n`=10^6, alors il y a 10^6 appels qui s'empilent, et qui se dépilent en calculant les multiplications $\longrightarrow$ stack overflow !

Récursivité terminale

$\longrightarrow$ par exemple, utilisation d'un accumulateur

Un exemple (avec récursivité terminale) :

```
let rec fact acc = function
  0 -> acc
| n -> fact (n * acc) (n-1) ;;

fact 1 10 ;; (* factorielle de 10 *)
```

$\longrightarrow$ OCaml comprend qu'il n'est pas obligé d'empiler tous les appels

démo

Plan

Champs mutables

Les champs des enregistrements peuvent être modifiés : mot-clé mutable.

```
type personne = { mutable age : int ; nom : string }
# let p = { age = 20 ; nom = "chimay" } ;;
val p : personne = {age = 20; nom = "chimay"}
# p.age <- 21 ;;
- : unit = ()
# p ;;
- : personne = {age = 21; nom = "chimay"}
# p.nom <- "chimay bleue" ;;
The record field label nom is not mutable
```

admirez l'effort pédagogique ...

Références


```
type 'a ref = {mutable contents : 'a}
```

Sorte de pointeurs sur une valeur quelconque.

- affectation : `( :=)`
- déréférencement : `( !)`

démo

Boucles


```
for nom = expr to expr do expr done
for nom = expr downto expr do expr done
while expr do expr done
```

démo

Plan

Plan