Programmation fonctionnelle en OCaml

Damien Rouhling (Lycée Victor Hugo)

10 mars 2022

Paradigme fonctionnel

- Sous paradigme du paradigme déclaratif.

Paradigme fonctionnel

- Sous paradigme du paradigme déclaratif.
- On déclare des fonctions (au sens mathématique).

Paradigme fonctionnel

- Sous paradigme du paradigme déclaratif.
- On déclare des fonctions (au sens mathématique).
- On écrit des expressions et non des instructions.

Paradigme fonctionnel

- Sous paradigme du paradigme déclaratif.
- On déclare des fonctions (au sens mathématique).
- On écrit des expressions et non des instructions.
- En particulier : on ne peut pas modifier le contenu d'une case mémoire.

Le langage OCaml

- Développé à Inria.

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).
  - Première version en 1987 (par Ascander Suarez).

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).
  - Première version en 1987 (par Ascander Suarez).
- En 1990, devient Caml light, basé sur un interprète de bytecode en C par Xavier Leroy.

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).
  - Première version en 1987 (par Ascander Suarez).
- En 1990, devient Caml light, basé sur un interprète de bytecode en C par Xavier Leroy.
- Étendu en Objective Caml (1996) puis renommé en OCaml (2011).

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).
  - Première version en 1987 (par Ascander Suarez).
- En 1990, devient Caml light, basé sur un interprète de bytecode en C par Xavier Leroy.
- Étendu en Objective Caml (1996) puis renommé en OCaml (2011).
  - Basé sur une amélioration de Caml light par Xavier Leroy.

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).
  - Première version en 1987 (par Ascander Suarez).
- En 1990, devient Caml light, basé sur un interprète de bytecode en C par Xavier Leroy.
- Étendu en Objective Caml (1996) puis renommé en OCaml (2011).
  - Basé sur une amélioration de Caml light par Xavier Leroy.
  - Contient des primitives pour la programmation orientée objet implémentées par Didier Rémy.

Le langage OCaml

- Développé à Inria.
- Appelé initialement Caml (Categorical Abstract Machine Language).
  - Dérivé de Standard ML (Meta Language).
  - Première version en 1987 (par Ascander Suarez).
- En 1990, devient Caml light, basé sur un interprète de bytecode en C par Xavier Leroy.
- Étendu en Objective Caml (1996) puis renommé en OCaml (2011).
  - Basé sur une amélioration de Caml light par Xavier Leroy.
  - Contient des primitives pour la programmation orientée objet implémentées par Didier Rémy.
- Caractéristiques : système de types puissant, gestion automatique de la mémoire, récursivité et fonctions d'ordre supérieur polymorphes.

Typage

- Tout est expression (y compris les fonctions).

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`.

  Exemple : `1`, `2 + 8`, `a / b`...

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`.

  Exemple : `1.`, `2.5 +. 3.2e5`, `a /. b`...

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`.

  Exemple : **true**, `b1 && b2`, `b1 || b2`, `not b`, `x = y`...

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`, `char`.

  Exemple : `'a'`, `'b'`...

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`, `char`, `string`.

  Exemple : `"a"`, `"bon" ^ "jour"`, ...

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`, `char`, `string`, `t1 * t2`.

  Exemple : `(1, 2) : int * int`, `(`**`true`**`, "abc") : bool * string`, `(1, 2, 3) : int * int * int`...

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`, `char`, `string`, `t1 * t2`, `unit`.

  Exemple : `()` (unique valeur, type utilisé pour les instructions)

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`, `char`, `string`, `t1 * t2`, `unit`.
- Les types sont inférés automatiquement et statiquement.

Typage

- Tout est expression (y compris les fonctions).
- Toute expression a une valeur et un type.
- Types de base : `int`, `float`, `bool`, `char`, `string`, `t1 * t2`, `unit`.
- Les types sont inférés automatiquement et statiquement.
- Le typage est fort.

Déclarations

- Un programme est une suite de déclarations.

Déclarations

- Un programme est une suite de déclarations.
- Exécuter un programme revient à évaluer les déclarations dans l'ordre.

Déclarations

- Un programme est une suite de déclarations.
- Exécuter un programme revient à évaluer les déclarations dans l'ordre.
- Syntaxe :

```
let <identifiant> = <expr>;;
```

Déclarations

- Un programme est une suite de déclarations.
- Exécuter un programme revient à évaluer les déclarations dans l'ordre.
- Syntaxe :

```
let <identifiant> = <expr>;;
```

- Possibilité de forcer le type :

```
let <identifiant> : <type> = <expr>;;
```

Déclarations

- Un programme est une suite de déclarations.
- Exécuter un programme revient à évaluer les déclarations dans l'ordre.
- Syntaxe :

```
let <identifiant> = <expr>;;
```

- Possibilité de forcer le type :

```
let <identifiant> : <type> = <expr>;;
```

- La liaison est statique :

```
let x = 3;;
let y = 5 + x;; (* ici, y vaut 8 *)
let x = 10;; (* ici, y vaut toujours 8 *)
```

Déclarations

- Un programme est une suite de déclarations.
- Exécuter un programme revient à évaluer les déclarations dans l'ordre.
- Syntaxe :

```
let <identifiant> = <expr>;;
```

- Possibilité de forcer le type :

```
let <identifiant> : <type> = <expr>;;
```

- La liaison est statique :

```
let x = 3;;
let y = 5 + x;; (* ici, y vaut 8 *)
let x = 10;; (* ici, y vaut toujours 8 *)
```

- Une expression particulière : la liaison locale.

```
let <identifiant> = <expr1> in <expr2>
```

Déclarations

- Un programme est une suite de déclarations.
- Exécuter un programme revient à évaluer les déclarations dans l'ordre.
- Syntaxe :

```
let <identifiant> = <expr>;;
```

- Possibilité de forcer le type :

```
let <identifiant> : <type> = <expr>;;
```

- La liaison est statique :

```
let x = 3;;
let y = 5 + x;; (* ici, y vaut 8 *)
let x = 10;; (* ici, y vaut toujours 8 *)
```

- Une expression particulière : la liaison locale.

```
let <identifiant> = <expr1> in <expr2>
```

- La déclaration simultanée est possible (aussi pour la liaison locale).

```
let <id1> = <expr1> and <id2> = <expr2>;;
```

Fonctions

- Syntaxe de la déclaration :

```
let <nom_fonction> <nom_argument> = <expr>;;
```

ou bien :

```
let <nom_fun> (<nom_arg> : <type_arg>) : <type_res>
  = <expr>;;
```

Fonctions

- Syntaxe de la déclaration :

```
let <nom_fonction> <nom_argument> = <expr>;;
```

ou bien :

```
let <nom_fun> (<nom_arg> : <type_arg>) : <type_res>
  = <expr>;;
```

- Possibilité de déclaration locale.

Fonctions

- Syntaxe de la déclaration :

  ```
  let <nom_fonction> <nom_argument> = <expr>;;
  ```

  ou bien :

  ```
  let <nom_fun> (<nom_arg> : <type_arg>) : <type_res>
    = <expr>;;
  ```

- Possibilité de déclaration locale.
- Fonctions anonymes : **function** `<nom_arg> -> <expr>`.
  On peut donc déclarer de manière équivalente :

  ```
  let <nom_fonction> = function <nom_arg> -> <expr>;;
  ```

Fonctions

- Syntaxe de la déclaration :

  ```
  let <nom_fonction> <nom_argument> = <expr>;;
  ```

  ou bien :

  ```
  let <nom_fun> (<nom_arg> : <type_arg>) : <type_res>
    = <expr>;;
  ```

- Possibilité de déclaration locale.
- Fonctions anonymes : **function** `<nom_arg> -> <expr>`.
  On peut donc déclarer de manière équivalente :

  ```
  let <nom_fonction> = function <nom_arg> -> <expr>;;
  ```

- Application : `<nom_fonction> <argument>`.
  Attention : `f x + 1` vs. `f (x + 1)`.

Fonctions à plusieurs arguments

Deux possibilités :

1. Un unique argument qui est un n-uplet.
2. Plusieurs arguments :

```
let <nom_fonction> <arg1> <arg2> ... <argn>
  = <expr>;;
```

Fonctions à plusieurs arguments

Deux possibilités :

1. Un unique argument qui est un n-uplet.
2. Plusieurs arguments :

```
let <nom_fonction> <arg1> <arg2> ... <argn>
  = <expr>;;
```

- La seconde version est dite curryfiée.

Fonctions à plusieurs arguments

Deux possibilités :

1. Un unique argument qui est un n-uplet.
2. Plusieurs arguments :

```
let <nom_fonction> <arg1> <arg2> ... <argn>
  = <expr>;;
```

- La seconde version est dite curryfiée.
- Elle permet des applications partielles :

```
let f x y = x + y;;
let g = f 1;;
let h = function y -> 1 + y;;
(* g et h sont égales *)
```

Fonctions à plusieurs arguments

Deux possibilités :

1. Un unique argument qui est un n-uplet.
2. Plusieurs arguments :

```
let <nom_fonction> <arg1> <arg2> ... <argn>
  = <expr>;;
```

- La seconde version est dite curryfiée.
- Elle permet des applications partielles :

```
let f x y = x + y;;
let g = f 1;;
let h = function y -> 1 + y;;
(* g et h sont égales *)
```

- On a donc des fonctions d'ordre supérieur.

Type des fonctions

- Type des fonctions de `t1` vers `t2` : `t1 -> t2`.
  Attention : `t1 -> t2 -> t3` vs. `(t1 -> t2) -> t3`.

Type des fonctions

- Type des fonctions de `t1` vers `t2` : `t1 -> t2`.
  Attention : `t1 -> t2 -> t3` vs. `(t1 -> t2) -> t3`.
- Polymorphisme : on peut écrire des fonctions valables pour n'importe quel type :

```
let id x = x;;
val id : 'a -> 'a = <fun>
```

Fonctions récursives

- On ajoute le mot-clé **rec**.

```
let f x = x + 1;;
let rec f y = if y = 0 then f 2 else 4 + y;;
```

Fonctions récursives

- On ajoute le mot-clé **rec**.

```
let f x = x + 1;;
let rec f y = if y = 0 then f 2 else 4 + y;;
```

- La récursivité croisée/mutuelle est possible via une déclaration simultanée.

Fonctions récursives

- On ajoute le mot-clé **rec**.

```
let f x = x + 1;;
let rec f y = if y = 0 then f 2 else 4 + y;;
```

- La récursivité croisée/mutuelle est possible via une déclaration simultanée.
- Teaser : récursivité terminale.

```
let somme n =
  let rec aux n acc =
    if n = 0 then acc else aux (n - 1) (acc + n)
  in aux n 0;;
```

Types construits

- Syntaxe de la définition de type :

```
type <nom_type> = <expr_type>;;
```

Types construits

- Syntaxe de la définition de type :

  ```
  type <nom_type> = <expr_type>;;
  ```

- Types enregistrements : semblables à des n-uplets dont les composantes seraient nommées.

Types construits

- Syntaxe de la définition de type :

```
type <nom_type> = <expr_type>;;
```

- Types enregistrements : semblables à des n-uplets dont les composantes seraient nommées.
- Types sommes :

```
type nombre_ou_rien =
  Rien | Entier of int | Reel of float;;
let x = Entier 2;;
```

Types construits

- Syntaxe de la définition de type :

```
type <nom_type> = <expr_type>;;
```

- Types enregistrements : semblables à des n-uplets dont les composantes seraient nommées.
- Types sommes :

```
type nombre_ou_rien =
  Rien | Entier of int | Reel of float;;
let x = Entier 2;;
```

- Les types peuvent être polymorphes :

```
type 'a option = None | Some of 'a;;
```

Types construits

- Syntaxe de la définition de type :

```
type <nom_type> = <expr_type>;;
```

- Types enregistrements : semblables à des n-uplets dont les composantes seraient nommées.
- Types sommes :

```
type nombre_ou_rien =
  Rien | Entier of int | Reel of float;;
let x = Entier 2;;
```

- Les types peuvent être polymorphes :

```
type 'a option = None | Some of 'a;;
```

- Types inductifs :

```
type 'a list = Nil | Cons of 'a * 'a list;;
```

Notations sur les listes : `[]` et `t :: q`.

Filtrage

- Un outil essentiel pour manipuler les éléments d'un type somme.

Filtrage

- Un outil essentiel pour manipuler les éléments d'un type somme.
- Syntaxe :

```
match <expr> with
| <motif> -> <expr>
| ...
| <motif> -> <expr>
```

Filtrage

- Un outil essentiel pour manipuler les éléments d'un type somme.
- Syntaxe :

```
match <expr> with
| <motif> -> <expr>
| ...
| <motif> -> <expr>
```

- Motifs : une constante, un nom, _, un constructeur appliqué à des motifs.

Filtrage

- Un outil essentiel pour manipuler les éléments d'un type somme.
- Syntaxe :

```
match <expr> with
| <motif> -> <expr>
| ...
| <motif> -> <expr>
```

- Motifs : une constante, un nom, _, un constructeur appliqué à des motifs.
- Motifs gardés : `<motif>` **when** `<expr_booléenne>`.

Bien plus encore

- Système d'exceptions.
- Programmation impérative (via le type `unit`).
- Programmation orientée objet.
- Système de modules et foncteurs.

Des questions ?