Chapitre 3 - Types fonctionnels

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251

Typage

Typage

OCaml dispose d'un typage **fort** et **statique** :

- Une variable a un seul type, qui ne change pas
- Pas de conversion implicite
- Le typage est vérifié *avant* l'exécution du programme
  - Si le programme compile, on n'aura pas d'erreur de type

Typage statique fort

Avantages :

- Erreurs détectées à la compilation plutôt qu'à l'exécution
- Programmes plus sûrs
- Programmes plus efficaces

Désavantages :

- Conversions explicites requises
- Verbosité plus élevée
- Erreurs de compilations parfois cryptiques

Erreur cryptique

Le programme (Haskell) suivant :

```
foldr (+) 0 1
```

Émet l'erreur de type :

```
<interactive>:1:1: error:
    • Could not deduce (Foldable t0)
        arising from a type ambiguity check for
        the inferred type for ‘it’
      from the context: (Foldable t, Num b, Num (t b))
        bound by the inferred type for ‘it’:
                   forall {t :: * -> *} {b}. (Foldable t, Num b, Num (t b)) => b
        at <interactive>:1:1-13
      The type variable ‘t0’ is ambiguous
      These potential instances exist:
        instance Foldable (Either a) -- Defined in ‘Data.Foldable’
        instance Foldable Maybe -- Defined in ‘Data.Foldable’
        instance Foldable ((,) a) -- Defined in ‘Data.Foldable’
        ...plus two others
        ...plus 26 instances involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In the ambiguity check for the inferred type for ‘it’
      To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
      When checking the inferred type
        it :: forall {t :: * -> *} {b}. (Foldable t, Num b, Num (t b)) => b
```

Types de base

Types primitifs

On a déjà vu :

- `int`
- `float`
- `bool`
- `char`
- `string`
- `unit`

Il existe aussi :

- `bytes` : pour manipuler des chaînes d'octets

Types composés

On a déjà vu :

- `'a list` : listes
- `'a * 'b`, `'a * 'b * 'c`, … : n-uplets
- `'a -> 'b` : fonction

Il existe aussi :

- `'a array` : tableau (impur)
- `'a option` : type optionnel
- `('a, 'b) result` : type “résultat” pour un calcul qui peut échouer

Variables de types (*type variables*)

`'a`, `'b` etc. sont des **variables de types** qui peuvent être remplacées par des types concrets.

Dans `'a list`, si

- `'a = int`, on a une liste d'entiers `int list`
- `'a = int list`, on a une liste de liste d'entiers `(int list) list`
- `'a = int -> int`, on a une liste de fonctions `(int -> int) list`

Types polymorphes

On peut laisser certains types abstraits, ce qui donne un type **polymorphe** :

- `'a list -> int` est une fonction qui prend une liste de *quelque chose* et retourne un entier
- `('a * 'b) list -> 'a` est une fonction qui prend une liste de paires, et retourne un élément du même type que les premiers éléments des paires

Polymorphisme en Java

Dans certains langages, on parle de *types génériques* plutôt que de polymorphisme, mais c'est la même chose

```
public static int getSize<T>(List<T> list);
```

est équivalent a :

```
let get_size (l : 'a list) : int = ...
```

Types composés : listes

- `'a list` est une liste dont les éléments sont de type `'a`
- Contient des éléments *homogènes* : tous de type `'a`
- La longueur d'une liste est variable
- `'a` est une **variable de type**

Types composés : n-uplets

- `'a * 'b` : une paire (2-uplet) dont le premier élément est de type `'a` et le second de type `'b`
- `'a * 'b * 'c` : un triplet
- Contient des éléments *hétérogènes* : les types des éléments peuvent être différents
- Longueur fixe

Attention : `int * int` est le **type** d'un double d'entiers, `(5, 3)` est une **valeur**

Types composés : fonctions

```
# not;;
- : bool -> bool = <fun>
```

- Une fonction est de type `'a -> 'b`
- Une fonction ne prend qu'un seul argument

Types composés : fonctions à plusieurs paramètres

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

- `'a -> 'b -> 'c` est équivalent à `'a -> ('b -> 'c)`
- C'est une fonction qui prend un argument et renvoie une fonction
- Le type d'une fonction est également appelé sa **signature**

Définition de types

Définition de types

On peut définir de nouveaux types avec le mot clé `type`

Synonyme de type

On peut définir un nouveau type comme égal à un type existant :

```
type point_2d = float * float
type file_path = string
```

Les types synonymes sont complètement équivalents à leurs définitions, donc interchangeables.

Équivalent au `typedef` en C :

Type énumération

Un type *énumération* a une liste de valeurs possibles, fixe :

```
type day = Monday | Tuesday | Wednesday | Thursday
         | Friday | Saturday | Sunday
type my_bool = False | True
```

- Le nom du type commence par une **majuscule**
- Les **constructeurs** possibles commencent par une **majuscule**

Type produit

Un constructeur peut prendre des paramètres avec of :

```
type three_ints = ThreeInts of int * int * int
```

Pour construire la valeur, on utilise le constructeur de type :

```
ThreeInts (1, 2, 3)
```

Type produit et synonyme de type

Attention, les deux types suivants sont différents :

```
type three_ints = ThreeInts of int * int * int (* type produit *)
type three_ints' = int * int * int (* type synonyme *)
```

On ne peut pas utiliser l'un à la place de l'autre

Type produit polymorphe

On peut définir des types produits qui dépendent de variables de type :

```
type ('a, 'b, 'c) triple = Triple of 'a * 'b * 'c
```

Le type est **paramétré** par d'autres types

Attention à bien distinguer :

- les paramètres de types `('a, 'b, 'c)`
- le type n-uplet `'a * 'b * 'c`

Type produit polymorphe

```
# type ('a, 'b, 'c) triple = Triple of 'a * 'b * 'c;;
type ('a, 'b, 'c) triple = Triple of 'a * 'b * 'c
# Triple (1, 'a', true);;
- : (int, char, bool) triple = Triple (1, 'a', true)
```

Type algébrique

On peut combiner les énumérations avec un type produit :

```
type 'a tree = Empty
             | Node of 'a tree * 'a * 'a tree
```

Types algébriques

Les types produits et sommes forment ce qu'on appelle les **types algébriques** Ils sont définis en terme de :

- somme (`A | B` représente $A \cup B$)
- produit (`A * B` représente $A \times B$)

Enregistrements (*records*)

On peut vite se retrouver avec des types complexes :

```
type configuration =
    string * (* User name *)
    string * (* Local host *)
    string * (* Remote host *)
    bool   * (* Is guest? *)
    bool   * (* Is superuser? *)
    string * (* Current directory *)
    string * (* Home directory *)
    int      (* Time connected *)
```

Enregistrements (*records*)

On peut définir des fonctions d'aides :

```
let username (username, _, _, _, _, _, _, _) = username
...
let is_guest (_, _, _, is_guest, _, _, _, _) = is_guest
...
```

Mais c'est pénible...

Enregistrements (*records*)

OCaml peut nous aider avec la **syntaxe d'enregistrements** (*record syntax*) :

```
type configuration = {
  username : string;
  local_host : string;
  remote_host : string;
  is_guest : bool;
  is_superuser : bool;
  current_dir : string;
  home_dir : string;
  time_connected : integer;
}
```

Enregistrements (*records*)

On peut construire un enregistrement et accéder à ses champs :

```
let c = {
  username = "foo";
  local_host = "127.0.0.1";
  ...
} in
c.username
```

Enregistrements (*records*)

Et on peut facilement mettre à jour des champ :

```
let change_dir (config : configuration) (new_dir : string) : configuration = {
  config with current_dir = new_dir
}
```

Typage de OCaml : résumé

- Types de bases (`bool`, `char`, `int`,…)
- Types composés :
  - Listes : `'a list`
  - Tuples : `'a * 'b`
  - Fonctions : `'a -> 'b`
- Types algébriques avec `type`
  - Somme : `X | Y`
  - Produit : `X of 'a * 'b`
- Constructions pratiques :
  - Synonymes
  - Enregistrements

Exemple : expressions mathématiques

```
type expr =
    | Plus of expr * expr          (* a + b *)
    | Minus of expr * expr         (* a - b *)
    | Times of expr * expr         (* a * b *)
    | Divide of expr * expr        (* a / b *)
    | Num of int

let example =
  (* 1 + (2 * 3) *)
  Plus ((Num 1), (Times ((Num 2), (Num 3))))
```

Filtrage par motif (*pattern matching*)

Filtrage par motif (*pattern matching*)

La forme `match ... with ...` permet de faire du filtrage par motif :

```
let rec expr_to_string (expr : expr) : string =
  match expr with
  | Plus (e1, e2) ->
    Printf.sprintf "%s + %s" (expr_to_string e1) (expr_to_string e2)
  | Minus (e1, e2) ->
    Printf.sprintf "%s - %s" (expr_to_string e1) (expr_to_string e2)
  | Times (e1, e2) ->
    Printf.sprintf "%s * %s" (expr_to_string e1) (expr_to_string e2)
  | Divide (e1, e2) ->
    Printf.sprintf "%s / %s" (expr_to_string e1) (expr_to_string e2)
  | Num n -> int_to_string n
```

Filtrage par motif (*pattern matching*)

On décrit la **forme** à satisfaire pour avoir une correspondance :

- Le caractère _ est un *joker* : correspond à n'importe quel motif (*wildcard*)
- Un constructeur de type correspond uniquement à ce constructeur
- Une valeur littérale correspond uniquement à cette valeur
- On peut ajouter des conditions avec `when`

```
let simplify (expr : expr) : expr =
  match expr with
  | Plus (Num 0, right) -> simplify right
  | _ -> expr
```

Filtrage par motif (*pattern matching*)

```
let rec simplify (e : expr) : expr =
  match e with
  | Add (left, Num 0) -> simplify left
  | Add (Num 0, right) -> simplify right
  | Mul (left, right) -> Mul (simplify left, simplify right)
  | _ -> e
```

Filtrage par motif (*pattern matching*)

La *forme* du motif vient généralement de la définition du type :

```
type month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec

let translate (month : month) : string =
  match month with
  | Jan -> "Janvier"
  | Feb -> "Février"
  | ...
```

Filtrage par motif (*pattern matching*)

Avec condition :

```
let sign (x : int) : int =
  match x with
  | x when x > 0 -> 1
  | x when x < 0 -> -1
  | 0 -> 0
```

Filtrage par motif (*pattern matching*)

On peut filtrer plusieurs motifs différents :

```
let rec simplify (expr : expr) : expr =
  match expr with
  | Plus (Num 0, e) -> simplify e
  | Sub (e, Num 0) -> simplify e
  | _ -> expr
```

Filtrage par motif (*pattern matching*)

```
let not (b : bool) : bool =
  match b with
  | true -> false
  | false -> true
```

Filtrage par motif (*pattern matching*)

```
let fst (pair : 'a * 'b) : 'a =
  match pair with
  | (x, _) -> x

let snd (pair : 'a * 'b) : 'b =
  match pair with
  | (_, y) -> y
```

Déconstruction de paramètres

Quand on s'attend à recevoir un paramètre d'une certaine forme, on peut le *déconstruire* directement :

```
let fst ((x, _) : 'a * 'b) : 'a = x
let snd ((_, y) : 'a * 'b) : 'b = y
```

Exemple : naturels de Peano

On peut définir les nombres naturels inductivement :

- 0 est un naturel
- si n est un naturel, $n + 1$ l'est aussi

```
type nat =
  | Zero
  | Succ of nat
```

Exemple : naturels de Peano

```
let zero = Zero
let one = Succ zero
let two = Succ one

let is_zero (n : nat) : bool =
  match n with
  | Zero -> true
  | Succ _ -> false

let inc (x : nat) : nat = Succ x
```

Exemple : naturels de Peano

```
let rec equal (n1 : nat) (n2 : nat) : bool =
  match (n1, n2) with
  | (Zero, Zero) -> true
  | (Succ x, Succ y) -> equal x y
  | _ -> false
```

Exemple : naturels de Peano

```
let rec add (n1 : nat) (n2 : nat) : nat =
  match n1 with
  | Zero -> n2
  | Succ pred_n1 -> add pred_n1 (Succ n2)
```

Exemple : naturels de Peano

```
let rec int_of_nat (n : nat) : int =
  match n with
  | Zero -> 0
  | Succ n -> 1 + (int_of_nat n)

let rec nat_of_int (n : int) : nat =
  match n with
  | 0 -> Zero
  | n when n > 0 -> Succ (nat_of_int (n - 1))
  | _ -> failwith "nat_of_int undefined on negative integers"
```

Le type `option`

Gestion des erreurs

Il y a trois façons de gérer les erreurs en OCaml :

- utiliser les exceptions
- utiliser le type `'a option`
- utiliser le type `'a result`

On préférera le type `option`.

Gestion des erreurs avec exceptions

On peut définir une nouvelle exception avec `exception`, et lever une exception avec `raise`

```
exception DivideByZero

let divide (x : int) (y : int) : int =
  match y with
  | 0 -> raise DivideByZero
  | Some y -> x / y
```

Gestion des erreurs avec exceptions

Avantage : c'est transparent, pas besoin de (trop) se soucier des cas d'erreur

Problème : c'est transparent, on ne peut pas être certain de l'absence d'exception

Définition de `option`

Défini dans la bibliothèque standard :

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

- Soit on a un résultat de type `'a`
- Soit on n'a rien

Gestion des erreurs avec `option`

```
let safe_divide (x : int) (y : int) : int option =
  match y with
  | 0 -> None
  | Some y -> Some (x / y)
```

option vs. null

En Java, on utilise parfois `null` pour dénoter l'absence d'une valeur.

Mais : au niveau du type, on ne sait pas si on doit potentiellement s'attendre à avoir un `null` ou pas.

```
public void foo(Object x) {
  println(x.toString()) // Va peut-être soulever une NullPointerException !
}
```

Erreurs dans la bibliothèque standard

Pour les fonctions standard qui peuvent échouer, on a généralement une version avec exception et une version avec `option`

```
# int_of_string;;
- : string -> int = <fun>
# int_of_string_opt;;
- : string -> int option = <fun>
```

→ On préférera les versions `_opt`

Erreurs dans la bibliothèque standard

```
# int_of_string "42";;
- : int = 42
# int_of_string "hello";;
Exception: Failure "int_of_string".

# int_of_string_opt "42";;
- : int option = Some 42
# int_of_string_opt "hello";;
- : int option = None
```

Paramètres optionnels avec `option`

Rappel :

```
# let f ?(x: int option) () = x;;
val f : ?x:int -> unit -> int option = <fun>
# f ~x:5 ();;
- : int option = Some 5
# f ();;
- : int option = None
```

- Si on a un `None`, aucune valeur n'a été passée pour le paramètre optionnel
- Si on a un `Some`, on a eu une valeur

Paramètres optionnels avec `option`

On aurait pu aussi utiliser une valeur par défaut :

```
# let f ?(x: int option = None) () = x;;
val f : ?x:int -> unit -> int option = <fun>
# f ~x:5 ();;
- : int option = Some 5
# f ();;
- : int option = None
```

Le type `list`

Liste simplement chaînée

Une liste est :

- soit vide
- soit composée d'un élément et d'un reste

En C :

```c
struct list {
    int val;
    struct list *next; // si NULL: c'est le dernier élément
};
```

Définition de liste en OCaml

Une liste est :

- soit vide
- soit composée d'un élément et d'un reste

```
type 'a list =
| Null
| Cons of 'a * 'a list
```

Définition de `list`

Définition réelle :

```
type 'a list = [] | (::) of 'a * 'a list
```

- Soit on a une liste vide : `[]`
- Soit on a un élément et un reste : `head :: tail`

Sucre syntaxique

```
1 :: 2 :: 3 :: []
```

est fastidieux à écrire... on a du sucre syntaxique pour nous aider :

```
[1; 2; 3]
```

Attention : le séparateur est ; (la virgule est réservée pour les n-uplets)

Manipulation de listes

Les définitions de fonctions sur des listes se font généralement par **filtrage par motif** et avec de la **récursion** :

- un cas de base (par exemple la liste vide)
- un cas récursif (par exemple pour la liste non vide)

Manipulation de listes

```
let rec length (l : 'a list) : int =
  match l with
  | [] -> 0
  | _ :: xs -> 1 + (length xs)
```

Manipulation de listes

```
let rec sum (l : int list) : int =
  match l with
  | [] -> 0
  | (x : xs) -> x + (sum xs)
```

Concaténation

```
let rec append (x : 'a list) (y : 'a list) =
  match x with
  | [] -> y
  | x :: xs -> x :: (append xs y)
```

On reconstruit la liste entièrement.

Requiert $\mathcal{O}(n)$ opérations où n est la longueur de la liste

Concaténation

Dans la bibliothèque standard, on a l'opérateur `@` pour concaténer :

```
# [1; 2] @ [3];;
- : int list = [1; 2; 3]
```

Répétitions

```
let rec replicate (x : 'a) (n : int) : 'a list =
  match n with
  | 0 -> []
  | n -> x :: (replicate x (n - 1))
```

Changer un élément

Exercice

Donner la définition de la fonction suivante :

```
val update : int -> 'a -> 'a list -> 'a list
```

De façon à ce que :

```
# update 3 'b' ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
['a'; 'b'; 'c'; 'b'; 'e'; 'f']
```

Filtrage et récursivité

On combine souvent filtrage par motif et récursivité en ayant :

- un (ou plusieurs) **cas de base** non récursif
- un (ou plusieurs) **cas récursif**

Filtrage et récursivité

Ici, la fonction qui garde tous les éléments d'une liste sauf le dernier; deux cas de base

```
let rec init (l : 'a list) : 'a list =
  match l with
  | [] -> failwith "init of empty list"
  | x :: [] -> []
  | x :: y :: l -> x :: (init y :: l)
```

Concepts clés

- **Variable de type** : 'a, utilisé pour des types **polymorphes**
- **Définition de nouveaux types** avec le mot clé `type`
- **Types algébriques** :
  - type somme/énumération : `type Foo = A | B`
  - type produit : `type Bar = X of int * string`
  - possibilité de définir des types polymorphes : `type 'a Baz = ...`
- **Enregistrements** : sucre syntaxique pour représenter des structures
- Le **filtrage par motif** se combine bien avec la définitions de nouveaux types
- Type `option` : soit `None`, soit `Some ...`
  - utilisé entre autre pour représenter des situations d'erreurs
- Type `list` : soit `[]`, soit `a :: b`

Exemple : génération procédurale

Exemple : génération procédurale

Voir dépôt d'exemples