Cours de programmation fonctionnelle et logique
Cours 5 : typage et polymorphisme

Table des matières

1 Notion de polymorphisme

polymorphe : étymologie = qui peut prendre plusieurs formes. Il s'agit d'objets ou de programmes qui peuvent servir *sans modifications* dans des contextes très différents.

Une fonction de tri qui s'applique à un seul type d'objets (e.g. les entiers) est *monomorphe*.

Une fonction de tri qui s'applique à tous les types d'objets est *polymorphe*. Cette fonction de tri pourra être employée sur des objets de types différents : pour trier un ensemble d'entiers ou un ensemble de chaînes de caractères. Mais, *a priori*, pas pour trier un ensemble contenant simultanément des entiers et des chaînes de caractères.

Des *objets* peuvent être polymorphes, comme la liste vide qui peut être vu que une liste d'entiers, ou de chaînes de caractères, etc. :

```
# [];;
 - : 'a list = []
```

1.1 Polymorphisme dans les expressions de type

L'introduction du polymorphisme nécessite une notion de type qui permette de remplacer plusieurs types différents¿ `ocaml` utilise des *paramètres de type* : `'a` (paramètre de type nommé “a”).

Le polymorphisme de `caml` est *paramétrique*. Il fonctionne en “tout ou rien” : un paramètre remplace n'importe quel type et non pas un certain nombre de types. Il n'y a pas de possibilités en `caml` de définir un type qui s'appliquerait uniquement à des entiers et à des chaînes de caractères. Il n'existe pas de type du style `(int or string) -> ...`

Un programme `caml` s'applique donc :

– soit à tous les types possibles (`'a -> ...`) ;
– soit à un seul et unique type (`int -> ...`).

1.2 Exemples simples

Fonction monomorphe (à cause de l'addition).

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

Fonction polymorphe (comme elle n'utilise pas son argument on peut l'appliquer à n'importe quoi).

```
#let fonction_un x = 1;;
 val fonction_un : 'a -> int = <fun>

#fonction_un 2;;
 - : int = 1

#fonction_un "oui";;
 - : int = 1
```

Une fonction polymorphe peut utiliser son argument.

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

Ici, *le type du résultat est exactement celui de l'argument.* Le paramètre de type `'a` remplace n'importe quel type, mais il remplace le même type partout dans l'expression de type.

```
#let liste_vide x = [];;
 val liste_vide : 'a -> 'b list = <fun>

#let liste_vide x =
   let tete = function t::r -> t | [] -> failwith "liste vide"
   in tete [];;
     val liste_vide : 'a -> 'b = <fun>
```

On peut, par exemple, utiliser la fonction `identite`, avec le type `string -> string`.

```
#identite "truc";;
 - : string = "truc"
```

Spécialisation : remplacement d'un paramètre de type par un type quelconque. Ici, `'a` est spécialisé en `string`. On n'est pas obligé de spécialiser un paramètre de type par un type de base. On peut, par exemple, spécialiser `'a` avec `int -> int`.

```
#identite successeur;;
 - : int -> int = <fun>

#identite successeur 3;;
 - : int = 4

#let reste = function
   t::r -> r
 |  []  -> failwith "liste vide";;
     val reste : 'a list -> 'a list = <fun>

#[];;
 - : 'a list = []

#reste [1];;
 - : int list = []
```

Ici, la spécialisation a eu lieu avec le type `int list`.

2 Synthèse du type le plus général

Inférence de type : le compilateur `caml` donne un type à chaque expression saisie, de manière totalement automatique. Pour ce faire, il utilise les types des *valeurs de base* et les *règles de typage* pour les constructions du langage.

Le type inféré contient le plus petit ensemble de contraintes nécessaires au bon déroulement de l'évaluation des expressions du programme. Plus petit ensemble de contraintes implique **type le plus général possible** pour chaque expression.

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

Son argument doit être entier car on lui ajoute l'entier 1 au moyen de l'addition des entiers de symbole `+`.

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

Il n'y a pas ici de contraintes sur l'argument de la fonction.

Le polymorphisme est une *conséquence de l'absence de contraintes* sur un argument ou une valeur.

```
#let double (f : int -> int) x = 2 * (f x);;
 val double : (int -> int) -> int -> int = <fun>
```

`x` doit être de type entier car on a imposé la contrainte que l'argument de `f` devait être de type entier : `f : int -> int`.

```
#let double f x = 2 * (f x);;
 val double : ('a -> int) -> 'a -> int = <fun>
```

La fonction est maintenant polymorphe. Le contrôleur de type a découvert que `f` avait pour seule contrainte de rendre un entier en résultat (car son résultat est argument de la multiplication des entiers), mais qu'il n'était nullement nécessaire que `f` prenne un entier en argument.

Comme le polymorphisme découle de l'absence de contraintes, un paramètre de type peut être remplacé sans risque d"erreurs par n'importe quel type, y compris par un type lui-même polymorphe.

```
#identite identite;;
 - : '_a -> '_a = <fun>
```

Le résultat de cette évaluation est la fonction `identite`.

3 Algèbre des types de `ocaml`

L'algèbre des types est l'ensemble des types utilisés par le système `ocaml`. Il existe trois catégories de types :

1. les types de bases : `int`, `string`, etc.
2. les types composites : `int -> int`, `int list`, etc.
3. les paramètres de type : `'a`.

Les types composites sont obtenus au moyen de constructeurs de types tel que `->`. Soient deux types $t1$ et $t2$. $t1$ `->` $t2$ est le type des fonctions ayant un argument de type $t1$ et rendant un résultat de type $t2$.

`->` est un constructeur de type binaire infixe. `list` et `array` sont des constructeurs de types unaires et postfixes. En `ocaml`, tous les constructeurs de types unaires sont postfixes. Les constructeurs de types sans arguments (`int`) sont dits *constructeurs de types constants*.

Les paires

Un nouveau constructeur de type binaire infixe : `*`. Soient deux types $t1$ et $t2$. $t1$ `*` $t2$ est le type produit cartésien des types $t1$ et $t2$, c'est-à-dire le type des couples dont le premier élément est de type $t1$ et le deuxième de type $t2$. La notation est identique à celle des mathématiques.

```
#(1,"e");;
 - : int * string = 1, "e"
```

Les paires peuvent être utilisées comme argument ou résultat d'une fonction.

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

#addition_soustraction (4,3);;
 - : int * int = 7, 1
```

Les paires permettent d'écrire des fonctions qui rendent plusieurs résultats. Les paires se généralisent aux n-uplets.

```
#(1, 2.3, "t",[5;6]);;
 - : int * float * string * int list = 1, 2.3, "t", [5; 6]
```

4 Curryfication

Une fonction qui prend une paire comme argument ne prend qu'un seul argument et non pas deux.

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

La fonction addition ne prend donc qu'un seul argument.

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

La fonction `add` prend deux arguments et est donc différente de la fonction `addition`.

Une fonction qui reçoit ses argument un par un, comme `add`, est dite **curryfiée**. Une fonction qui reçoit tous ses arguments à la fois sous la forme d'un n-uplet est dite **non curryfiée**. Le terme *curryfié* vient du nom du logicien Haskell Curry.

Application partielle

La différence entre les fonctions `add` et `addition` provient de la manière de les appliquer : il est légal d'appliquer `add` à un seul argument.

```
#let suivant = add 1;;
 val suivant : int -> int = <fun>

#suivant 3;;
 - : int = 4
```

Au contraire, la fonction `addition` doit recevoir tous ses arguments en une seule fois.

Application partielle : c'est la capacité qu'a une fonction curryfiée de ne recevoir qu'un certain nombre de ses arguments.

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

Les fonctions `add` et `sum` sont équivalentes.

Curryfication et "type flèche"

Une fonction curryfiée est un cas particulier de fonctionnelle puisqu'elle permet de créer d'autres fonctions, en fixant un certain nombre de ses arguments. Cette propriété est inscrite dans le type de la fonction curryfiée :

```
#add;;
 - : int -> int -> int = <fun>
```

Le constructeur -> associe à droite. Donc `add` est de type : `int -> (int -> int)`. Autrement dit, `add` est une fonction qui prend comme argument un entier et qui renvoie une objet de type `int -> int`, c'est-à-dire une fonction qui prend en entrée un entier et qui renvoie un entier.

```
#let ajoute_deux = add 2;;
 val ajoute_deux : int -> int = <fun>

#add 2 3;;
 - : int = 5

#(add 2) 3;;
 - : int = 5

#ajoute_deux 3;;
 - : int = 5
```

Autre approche

Pour tout entier `n` on peu écrire une fonction

```
#let n = 732;;
 val n : int = 732

#let ajoute_n x = n + x;;
 val ajoute_n : int -> int = <fun>
```

`add` est une fonction générique qui généralise toutes ces fonctions : c'est une fonctionnelle.

Pour une fonction curryfiée très générale, spécialiser un de ses arguments peut permettre d'obtenir une fonction intéressante. Une fonctionnelle de tri acceptant la fonction de comparaison en premier paramètre peut être spécialisée par application partielle en fonction de tri croissant ou de tri décroissant.

Fonctionnelle de curryfication et de dé-curryfication

Fonctionnelle qui associe à une fonction non curryfiée acceptant une paire en argument, la version curryfiée correspondante.

```
#let curry f = function x -> function y -> f (x,y);;
 val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Fonctionnelle inverse :

```
#let uncurry f = function (x,y) -> f x y;;
 val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

5 Algorithme de typage

5.1 Unicité du type

Dans une fonction, toutes les occurences d'un même argument sont du même type.

```
#let f x = 3;;
 val f : 'a -> int = <fun>

#f 4;;
 - : int = 3

#f "a";;
 - : int = 3

#(f 4) + (f "a");;
 - : int = 6
```

```
#let g = f in (g 4) + (g "a");;
 - : int = 6
```

```
#let g = (function x -> (f x) + 1 ) in (g 4) + (g "a");;
 - : int = 8
```

```
#let g h = (h 4) + (h "a");;
 Characters 21-24:
This expression has type string but is here used with type int
```

5.2 Premier exemple

```
#let double f x = 2*(f x);;
 val double : ('a -> int) -> 'a -> int = <fun>
```

```
f : a
x : b
2*(f x) : c
double : a -> b -> c
```

`*` : opération primitive, donc de type connu `int -> int -> int`

```
2 : int = vrai
(f x) : int
c : int
(f x) : int
f : d -> e, a = d -> e
x : d, b = d
e = int
a -> b -> c
(d -> e) -> b -> c
(d -> e) -> d -> c
(d -> int) -> d -> int
```

d'où le type.

5.3 Deuxième exemple

```
let machin n f x = f (n f x);;
n : a
f : b
x : c
f (n f x) : d
machin : a -> b -> c -> d
```

`f` est appliquée à un argument, donc c'est une fonction.

```
f : e -> d, b = e -> d
(n f x) : e
```

`n` est appliquée (premier élément d'un groupe entre parenthèses), donc c'est une fonction.

```
n : b -> c -> e = a
machin : a -> b -> c -> d
machin : (b -> c -> e) -> b -> c -> d
machin : ((e -> d) -> c -> e) -> (e -> d) -> c -> d
```

```
#let machin n f x = f (n f x);;
 val machin : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c -> 'b = <fun>
```

5.4 Typage avec contraintes

```
#let machin (n: ('a->'a)->('a->'a)) f x = f (n f x);;
 val machin : (('a -> 'a) -> 'a -> 'a) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

```
n : (a->a)->(a->a)
f : b
x : c
f (n f x) : d
machin : ((a->a)->(a->a)) -> b -> c -> d
f : e -> d = b
n f x : e
n : b -> c -> e = (a->a)->(a->a)
b -> c -> e = b -> (c -> e)
b = a -> a = e -> d, e = a, d = a
c -> e = a -> a, c = a, e = a
machin : ((a->a)->(a->a)) -> (a->a) -> a -> a
```

5.5 Algorithme général

Algorithme

Initialisation

- $\Gamma = \emptyset$: ensemble des déclarations de types de variables ;
- $E = \emptyset$: système d'équations entre les types ;
- $T =$ `t :a` : ensemble des expressions à typer.

Étape courante

On prend une expression `u :a` de T et on l'enlève T.

- `u = function x -> v`.
  - $\Gamma = \Gamma \cup \{$`x :b`$\}$, où `b` est une nouvelle lettre.
  - On rajoute `v :c` dans T, où `c` est une nouvelle lettre.
  - $E = E \cup \{$`a = b-> c`$\}$.
- `u = x` où `x` est une variable. Alors il existe un type `b` tel que `x :b` $\in \Gamma$. $E = E \cup \{$`a = b`$\}$.
- `u = (v w)`. Soit `b` est une nouvelle lettre. $T = T \cup \{$`v : b -> a`$\} \cup \{$`w : b`$\}$.
- `u = if v then w else z`. $T = T \cup \{$`v : bool`$\} \cup \{$`w : a`$\} \cup$ `z : a`.

Exemple

```
#let double = function f -> function x -> 2*(f x);;
 val double : ('a -> int) -> 'a -> int = <fun>
```

Initialisation

- $\Gamma = \{$`2 : int`, `* : int -> int -> int`$\}$;
- $E = \emptyset$;
- $T =$ `function f -> function x -> 2*(f x) : a`.

Étapes successives

1. `u = function f -> function x -> 2*(f x) : a`, $T = \emptyset$.
   - $\Gamma = \Gamma \cup \{$`f : b`$\}$.
   - $T = \{$`function x -> 2*(f x) : c`$\}$.
   - $E = E \cup \{$`a = b -> c`$\}$.
2. `u = function x -> 2*(f x) : c`, $T = \emptyset$.
   - $\Gamma = \Gamma \cup \{$`x : d`$\}$.
   - $T = \{$`2*(f x) : e`$\}$.
   - $E = E \cup \{$`c = d -> e`$\}$.
3. `u = 2*(f x) : e`, $T = \emptyset$.
   - $T = \{$`2 : g`, `(f x) : h`, `* : g -> h -> e`$\}$.
4. `u = 2 : g`, $T = \{$`(f x) : h`, `* : g -> h -> e`$\}$.
   - `2 : int` $\in \Gamma \Rightarrow E = E \cup \{$`g = int`$\}$.
5. `u = * : g -> h -> e`, $T = \{$`(f x) : h`$\}$.
   - `* : int -> int -> int` $\in \Gamma \Rightarrow E = E \cup \{$`g = int`, `h = int`, `e = int`$\}$.
6. `u = (f x) : h`, $T = \emptyset$.
   - $T = \{$`f : i -> h`, `x : i`$\}$.
7. `u = f : i -> h`, $T = \{$`x : i`$\}$.
   - $E = E \cup \{$`b = i -> h`$\}$.
8. `u = x : i`, $T = \emptyset$.
   - $E = E \cup \{$`d = i`$\}$.

Résultat : $E = \{$`a = b -> c`, `c = d -> e`, `g = int`, `g = int`, `h = int`, `e = int`, `b = i -> h`, `d = i`$\}$

```
a = b -> c
a = (i -> h) -> c
a = (i -> int) -> (d -> e)
a = (i -> int) -> (i -> int)
double : ('a -> int) -> 'a -> int
```