Introduction aux types

évaluation d'une expression $\Longrightarrow$ type et valeur

```
# 10 + 4;;
- : int = 14
```

Un type est un ensemble de valeurs.

Exemples:

type Booléen `bool = { true, false }`

type Caractères `char = { 'a', 'A', ...}`

Un certain nombre d'opérateurs (fonctions) sont prédéfinis pour les types prédéfinis.

Les fonctions OCaML sont typées: elles s'appliquent à des arguments ayant chacun un type défini et retournent une valeur d'un type également défini.

Si on utilise une fonction avec au moins un argument n'ayant pas le bon type, l'évaluation s'arrête à la compilation avec une erreur et la fonction n'est pas appelée.

Un type avec un ensemble d'opérateurs s'appliquant sur les valeurs d'un ensemble de types peut être appelé un type abstrait.

Inférence et vérification de types

à la compilation, OCaML infère le type de chaque expression ⇒
détection de certaines erreurs
si échec ⇒ pas d'évaluation
infère le type le plus général possible (variables de type `'a, 'b, ...`)

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# let f x = x, x;;
val f : 'a -> 'a * 'a = <fun>
# let f x y = x, y;;
val f : 'a -> 'b -> 'a * 'b = <fun>
# let f x = x + 1;;
val f : int -> int = <fun>
# let f x = x, x;;
val f : 'a -> 'a * 'a = <fun>
# let f x = x ^ x;;
val f : string -> string = <fun>
# let f x = x ^ (x + 1);;
Error: This expression has type string but an expression
       was expected of type int
```

Opérateurs de types

Les types peuvent être combinés à l'aide d'opérateurs de types pour obtenir de nouveaux types.
Exemples

- type contenant les couples constitués d'un entier et d'un flottant
- type des fonctions des entiers vers les booléens

Opérateur de fonction

L'opérateur de type `->` permet d'obtenir le type d'une fonction d'une variable d'un type vers un autre type

```
# let carre x = x * x;;
  val carre : int -> int = <fun>
# sin;;
- : float -> float = <fun>
```

`carre` fonction prend en paramètre un entier, retourne un entier
`sin` prend en paramètre un flottant, retourne un flottant

Parenthésage

Le parenthésage par défaut d'une séquence d'opérateurs `->` est de droite à gauche `a1 -> a2 -> ... an` est équivalent à `a1 -> (a2 -> (... -> (an-1-> an)...)` contrairement à l'appel de fonction qui est de gauche à droite.

```
# max;;
- : 'a -> 'a -> 'a = <fun>
# max 3;;
- : int -> int = <fun>
# max 3 4;;
- : int = 4
```

`max` est une fonction de deux arguments, `max 3` est une fonction d'un argument entier et fera le max entre cet argument et 3. `max 3 4` est un entier.

- Exemple de la fonction `compose f g`

Opérateur de produit cartésien (couples)

L'opérateur `*` est l'opérateur de produit cartésien. Le produit cartésien de A et de B est l'ensemble:

$$A \times B = \{(a, b) \mid a \in A \text{ et } b \in B\}$$

Il permet de représenter l'ensemble des tuples d'éléments appartement respectivement à un type donné. Exemples: `int * int`, `int * float * int`

```
# 2, 3;;
  - : int * int = (2, 3)
# fst (2, 3)
- : int 2
# snd (2, 3)
- : int 3
# 2, 3.5, 'a';;
- : int * float * char = (2, 3.5, 'a')
```

À noter les accesseurs `fst` et `snd`.

```
# let couple_square x = x, x *x;;
val couple_square : int -> int * int = <fun>
# let pair x = x, x;;
val pair : 'a -> 'a * 'a = <fun>
```

À noter dans le dernier exemple l'apparition de `'a` qui est une variable pouvant représenter un type quelconque. En effet, le corps de la fonction permet d'inférer que le résultat est un couple mais ne permet pas d'obtenir le type de `x`. Nous verrons plus loin cette notion de type paramétré par une variable de type.

Produit cartésien généralisé (tuples)

Le produit cartésien binaire se généralise aux tuples.

$$E_1 \times E_2 \times \cdots \times E_n = \{(e_1, e_2, \cdots, e_n) \mid e_i \in E_i \forall i, 1 \leq i \leq n\}$$

Exemples:

```
# 1, 2, 3;;
- : int * int * int = (1, 2, 3)
# (1, 2, 3);;
- : int * int * int = (1, 2, 3)
# 1, (2, 3);;
- : int * (int * int) = (1, (2, 3))
# (1, 2), 3;;
- : (int * int) * int = ((1, 2), 3)
```

Les tuples peuvent être paramètres des fonctions.

```
# let s3 (i, j, f) = i + j + int_of_float f;;
val s3 : int * int * float -> int = <fun>
# s3 (1, 2, 3.);;
- : int = 6
```

Une fonction peut retourner un tuple:

Filtrage

On peut récupérer les valeurs d'un tuple par filtrage:

```
# let tuple = 1, "deux", 3.5;;
val tuple : int * string * float = (1, "deux", 3.5)
# let i, str, f = tuple in i + int_of_float f, str;;
- : int * string = (4, "deux")
```

Requête `type`

requête `type` permet de donner un nom à un type (en général composé)

```
# type point2D = Point of float * float;;
type point2D = Point of float * float
# Point(1.2, 3.4);;
- : point2D = Point (1.2, 3.4)
```

`point2D` est le nom de type choisi. `Point` est le nom de constructeur choisi pour représenter un point. `type`, `of`, `float` sont prédéfinis en OCaML.
Le séparateur `|` correspond à une union (ou somme) de types.

```
# type int_or_infinity = Int of int | Infinity;;
type int_or_infinity = Int of int | Infinity
```

Des valeurs de ce type sont: `Int 5`, `Int (-2)`, `Infinity`.

```
type int_or_infinity = Int of int | Infinity
# let div n d =
  if d = 0 then
    if n = 0 then failwith "undefined form 0/0"
    else Infinity
  else Int(n/d);;
val div : int -> int -> int_or_infinity = <fun>
# div 3 0;;
- : int_or_infinity = Infinity
# div 3 2;;
- : int_or_infinity = Int 1
# div 0 0;;
Exception: Failure "undefined form 0/0".
# type form = Square of float | Circle of float | Rectangle
type form = Square of float | Circle of float | Rectangle o
# Rectangle (10., 3.);;
- : form = Rectangle (10., 3.)
# Square(3.4);;
- : formes = Square 3.4
```

La construction `match`

La construction `match` permet d'inspecter la forme d'une valeur et de récupérer parties de la valeur dans des variables locales.

```
# let perimeter_rect w h = 2. *. (w +. h);;
val perimeter_rect : float -> float -> float = <fun>
# let perimeter_circle r = 2. *. 3.14 *. r;;
val perimeter_circle : float -> float = <fun>
# let perimeter form =
  match form with
    Rectangle(w, h) -> perimeter_rect w h
  | Circle r -> perimeter_circle r
  | Square c -> perimeter_rect c c;;
val perimeter : formes -> float = <fun>
```