Les types

Un type t est défini *récursivement* par les cas :

- simple : char, bool, int, string, float, unit
- produit cartésien : $t_1 \times \ldots \times t_n$
- fonctionnel : $t_1 \rightarrow t_2$
- parenthésé : (t)
- variable libre : α, β, γ etc.
- type paramétré : α list

Remarque

Jusqu'à présent, nous n'avons pas rencontré de valeurs de type float, char ou string.

Règles syntaxiques sur les types

- Nous notons $\times$ ce qui s'écrit * en ascii.
- Nous notons α, β etc. ce qui s'écrit respectivement 'a, 'b etc. en ascii.
- Le produit cartésien est n-aire, et non binaire comme en mathématiques, car $\times$ n'est pas associatif en OCaml :
  $t_1 \times t_2 \times t_3 \neq (t_1 \times t_2) \times t_3 \neq t_1 \times (t_2 \times t_3)$
- La flèche est utilisée aussi dans les expressions. Elle associe à droite : $t_1 \to t_2 \to \ldots \to t_n$ équivaut à
  $t_1 \to (t_2 \to (\ldots (t_{n-1} \to t_n)) \ldots)$
- Le produit cartésien est prioritaire sur la flèche :
  $t_1 \times t_2 \to t_3$ équivaut à $(t_1 \times t_2) \to t_3$

Types, constantes simples et primitives

Les compilateur associe un type à chaque expression du programme : on parle d'*inférence de types statique*. Pour les constantes simples, nous avons

unit	()	
bool	**true false**	`&&` `||` `not`
int	1 2 max_int etc.	`+` `-` `*` `/` etc.
float	1.0 2. 1e4 etc.	`+.` `-.` `*.` `/.` `cos` etc.
char	'a' '\n' '\097' etc.	Char.code Char.chr etc.
string	"a\tb\010c\n" etc.	`^` s.[i] s.[i] ← c etc.

Les opérations sur les flottants sont notées différemment de leurs homologues sur les entiers. Ce que nous notons joliment ← s'écrit <- en ascii.

L'évaluation des opérateurs booléens

- Les opérateurs booléens sont *séquentiels*, c.-à-d. qu'ils n'évaluent leurs arguments que si c'est nécessaire, l'évaluation se faisant de la gauche vers la droite.

Extension de la syntaxe des types et des phrases

On étend la syntaxe des phrases pour permettre de lier un type à un nom, comme on peut le faire pour les expressions.

- liaison de type (ou alias) type $q = t$;;

où q dénote une variable de type (commençant par une minuscule).

- types récursifs type $q_1 = t_1$ [**and** $q_2 = t_2$ ...] ;;

Pour utiliser ces variables, il faut étendre la syntaxe des types :

- variable q

On peut maintenant écrire

```
type abscisse = float;;
type ordonnée = float;;
type point = abscisse * ordonnée;;
```

Inférence de types

Les n-uplets sont homogènes et leur arité est fixée par leur type :

une paire	(1,2)	de type	int $\times$ int
et un triplet	(1,2,3)	de type	int $\times$ int $\times$ int

sont incompatibles.

```
# let milieu x y = (x+y)/2;;
val milieu : int → int → int
# let milieu (x,y) = (x+y)/2;;
val milieu : int × int → int
```

Polymorphisme

n-uplets

Les projections sont polymorphes sur les *n*-uplets de *même arité* :

$$\textbf{fun}(x, y, z) \to x \quad \text{a pour type} \quad (\alpha \times \beta \times \gamma) \to \alpha$$

Fonction puissance

```
# let rec power f n =
    if n <= 0 then fun x -> x
              else compose f (power f (n-1));;
```

val power : $(\alpha \to \alpha) \to$ int $\to (\alpha \to \alpha)$ = $\langle$fun$\rangle$

Polymorphisme (suite)

```
# let compose f g = fun x -> f (g (x));;
```
val compose : $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta = \langle fun \rangle$

Le type de la fonction compose se construit ainsi :

- le premier argument f est une fonction quelconque, donc de type $\alpha \to \beta$;
- le second argument g est une fonction dont le résultat doit être passé en argument à f, donc de type α ;
- le domaine de g est quelconque, donc g est de type $\gamma \to \alpha$;
- la fonction compose prend un argument `x` qui doit être passé à g, donc du type γ ; finalement, le résultat de compose est retouné par f, donc de type β.

Égalité structurelle

L'opérateur d'égalité est polymorphe et ne peut être défini en OCaml :

```
# ( = );;
- : α → α → bool = ⟨fun⟩
```

Donc attention à ce qu'il coïncide avec *votre* notion d'égalité.
C'est l'égalité mathématique : deux valeurs sont égales si elles ont la même structure et si leurs parties respectives sont égales. Ne marche pas avec les expressions fonctionnelles (problème indécidable).

```
# 1 = 1 && "oui" = "oui";;
- : bool = true
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
```

On note <> la négation de l'égalité.