Définition de nouveaux types

Lycée Henri Poincaré - Nancy

17 février 2021

Sommaire

Sommaire

Abréviations (alias)

Exemple :

```
# type couple = int * int;;
type couple = int * int
```

Exercice

Écrire une fonction `swap: couple -> couple` qui prend en entrée un couple `(c1,c2)` et renvoie le couple `(c2,c1)`.

Solution de l'exercice

```
# let swap (c: couple): couple =
    let c1,c2 = c in
    c2,c1;;
val swap : couple -> couple = <fun>
```

Attention, si on ne force pas le type :

```
# let swap c =
    let c1,c2 = c in
    c2,c1;;
val swap : 'a * 'b -> 'b * 'a = <fun>
```

Sommaire

Type somme

- Type somme = objets définis par disjonction de cas
- Syntaxe générale :

```
type <nom du type> =
   | <Nom constructeur 1>
   | <Nom constructeur 2>
   | <Nom constructeur 3>
     ...;;
```

- Le nom du type commence par une minuscule
- Le nom d'un constructeur commence par une majuscule

Type somme

- Type somme = objets définis par disjonction de cas
- Syntaxe générale :

```
type <nom du type> =
    | <Nom constructeur 1>
    | <Nom constructeur 2>
    | <Nom constructeur 3>
      ...;;
```

- Le nom du type commence par une minuscule
- Le nom d'un constructeur commence par une majuscule

Exercice

- Définir un type `couleur_primaire` pour représenter les couleurs bleu, rouge et vert.
- Écrire une fonction `est_bleu: couleur_primaire -> bool` qui renvoie `true` si la couleur en paramètre est le bleu.

Solution de l'exercice

```
type couleur_primaire =
  | Bleu
  | Rouge
  | Vert;;
```

```
let est_bleu (c: couleur_primaire): bool = match c with
  | Bleu -> true
  | _    -> false;;
```

Type somme

- Un constructeur peut prendre un argument :

```
type <nom du type> =
  | <Nom constructeur 1> of <type>
  | <Nom constructeur 2> of <type>
  | <Nom constructeur 3> of <type>
    ...;;
```

Type somme

- Un constructeur peut prendre un argument :

```
type <nom du type> =
  | <Nom constructeur 1> of <type>
  | <Nom constructeur 2> of <type>
  | <Nom constructeur 3> of <type>
    ...;;
```

Exercice

- Écrire un type `nombre` permettant de représenter un entier ou bien un réel
- Écrire une fonction `somme: nombre -> nombre -> nombre` qui additionne deux nombres
- Utilisez la fonction `somme` pour sommer `4` et `0.1`.

Solution de l'exercice

```
type nombre =
  | Entier of int
  | Reel of float;;
```

```
let somme (n: nombre) (m: nombre): nombre =
  match n,m with
  | Entier x, Entier y -> Entier (x+y)
  | Reel   x, Reel   y -> Reel (x+.y)
  | Entier x, Reel   y -> Reel (float_of_int x +. y)
  | Reel   x, Entier y -> Reel (x +. float_of_int y);;
```

```
# let n = Entier 4 and m = Reel 0.1 in
    somme n m;;
- : nombre = Reel 4.1
```

Type somme

Un type somme peut être récursif

Exercice

- Définir un type `liste_entiers` pour représenter des listes chaînées d'entiers.
- Écrire une fonction `max: liste_entiers -> int` qui calcule le maximum des éléments de la liste.
- Tester la fonction `max` avec la liste qui contient les éléments `2` et `3`.

Solution de l'exercice

```
type liste_entiers =
  | Vide
  | Cons of int * liste_entiers;;
```

```
let rec max (li: liste_entiers): int = match li with
  | Vide           -> failwith "liste vide"
  | Cons(e, Vide)  -> e
  | Cons(e,li2)    -> let m = max li2 in
                      if e < m then m else e;;
```

```
let li = Cons(2, Cons(3,Vide)) in
    max li;;
```

Type somme

- Type somme polymorphe :

```
1  type 'a liste =
2    | Vide
3    | Cons of 'a * 'a liste;;
```

Type somme

- Type somme polymorphe :

```
type 'a liste =
  | Vide
  | Cons of 'a * 'a liste;;
```

- Type construit = type somme avec un seul constructeur :

```
type nom =
  | Nom of string;;
```

Type somme

- Type somme polymorphe :

```
type 'a liste =
  | Vide
  | Cons of 'a * 'a liste;;
```

- Type construit = type somme avec un seul constructeur :

```
type nom =
  | Nom of string;;
```

- Type constant = type construit avec constructeur sans argument :

```
type noneType =
  | None;;
```

Sommaire

Type produit (ou type enregistrement)

- Type produit = produit cartésien
- Déclarer un type produit :

```
type <nom type> = {
    <nom champ 1>: <type champ 1>;
    <nom champ 2>: <type champ 2>;
    ...
  };;
```

Type produit (ou type enregistrement)

- Type produit = produit cartésien
- Déclarer un type produit :

```
type <nom type> = {
    <nom champ 1>: <type champ 1>;
    <nom champ 2>: <type champ 2>;
    ...
  };;
```

- Déclarer une variable :

```
let <nom variable> = {
    <nom champ 1> = <valeur champ 1>;
    <nom champ 2> = <valeur champ 2>;
    ...
  };;
```

Type produit (ou type enregistrement)

- Type produit = produit cartésien
- Déclarer un type produit :

```
type <nom type> = {
    <nom champ 1>: <type champ 1>;
    <nom champ 2>: <type champ 2>;
    ...
  };;
```

- Déclarer une variable :

```
let <nom variable> = {
    <nom champ 1> = <valeur champ 1>;
    <nom champ 2> = <valeur champ 2>;
    ...
  };;
```

- Accéder à un champ :

```
<nom variable>.<nom champ>
```

Exercice

Exercice

- Définir un type `cplx` pour représenter les nombres complexes.
- Écrire une fonction `somme: cplx -> cplx -> cplx` qui fait la somme de deux complexes.
- Testez la fonction `somme` avec $1 + i$ et $0.5 - 2i$

Exercice

Exercice

- Définir un type `cplx` pour représenter les nombres complexes.
- Écrire une fonction `somme: cplx -> cplx -> cplx` qui fait la somme de deux complexes.
- Testez la fonction `somme` avec $1 + i$ et $0.5 - 2i$

```
1  type cplx = {re: float; im: float;};;
```

Exercice

Exercice

- Définir un type `cplx` pour représenter les nombres complexes.
- Écrire une fonction `somme: cplx -> cplx -> cplx` qui fait la somme de deux complexes.
- Testez la fonction `somme` avec $1 + i$ et $0.5 - 2i$

```
type cplx = {re: float; im: float;};;
```

```
let somme (z1: cplx) (z2: cplx): cplx = {
    re = z1.re +. z2.re;
    im = z1.im +. z2.im;
  };;
```

Exercice

Exercice

- Définir un type `cplx` pour représenter les nombres complexes.
- Écrire une fonction `somme: cplx -> cplx -> cplx` qui fait la somme de deux complexes.
- Testez la fonction `somme` avec $1 + i$ et $0.5 - 2i$

```
type cplx = {re: float; im: float;};;
```

```
let somme (z1: cplx) (z2: cplx): cplx = {
    re = z1.re +. z2.re;
    im = z1.im +. z2.im;
  };;
```

```
# let z1 = {re = 1. ; im = 1. ;}
  and z2 = {re = 0.5; im = -2.;} in
    somme z1 z2;;
- : cplx = {re = 1.5; im = -1.}
```

Type produit (ou type enregistrement)

Champs mutables (modifiables)

Pour définir un champ mutable :

```
type <nom type> = {
    mutable <nom champ 1>: <type champ 1>;
    mutable <nom champ 2>: <type champ 2>;
    ...
  };;
```

Pour changer valeur d'un champ mutable :

```
<nom variable>.<nom champ> <- <nouvelle valeur champ>
```

Type produit (ou type enregistrement)

Champs mutables (modifiables)

Pour définir un champ mutable :

```
type <nom type> = {
    mutable <nom champ 1>: <type champ 1>;
    mutable <nom champ 2>: <type champ 2>;
    ...
  };;
```

Pour changer valeur d'un champ mutable :

```
<nom variable>.<nom champ> <- <nouvelle valeur champ>
```

Exercice

- Définir un type `eleve` avec les champs `nom`, `prenom` et `age`.
- Écrire une fonction `anniversaire: eleve -> unit` qui ajoute un an à l'élève.

Solution de l'exercice

```
type eleve = {
    nom: string;
    prenom: string;
    mutable age: int
  };;
```

```
let anniversaire (e: eleve): unit =
  e.age <- e.age + 1;;
```

Type produit (ou type enregistrement)

- Type produit polymorphe :

```
1  type ('a,'b) couple = {
2       fst: 'a;
3       snd: 'b;
4     };;
```

Type produit (ou type enregistrement)

- Type produit polymorphe :

```
type ('a,'b) couple = {
    fst: 'a;
    snd: 'b;
  };;
```

- Type produit récursif :

```
type personne = {
    nom: string;
    enfants: personne list;
  };;
```

Sommaire

Exceptions

- Pour indiquer une erreur lors d'une exécution
- Exemple :

```
# 1/0;;
Exception: Division_by_zero.
```

Exceptions

- Pour indiquer une erreur lors d'une exécution
- Exemple :

```
# 1/0;;
Exception: Division_by_zero.
```

- Type d'une exception : `exn`

```
# Division_by_zero;;
- : exn = Division_by_zero
```

Exceptions

- Pour indiquer une erreur lors d'une exécution
- Exemple :

```
# 1/0;;
Exception: Division_by_zero.
```

- Type d'une exception : `exn`

```
# Division_by_zero;;
- : exn = Division_by_zero
```

- Définir une exception

```
exception <Nom de l'exception>
```

Exceptions

- Pour indiquer une erreur lors d'une exécution
- Exemple :

```
# 1/0;;
Exception: Division_by_zero.
```

- Type d'une exception : `exn`

```
# Division_by_zero;;
- : exn = Division_by_zero
```

- Définir une exception

```
exception <Nom de l'exception>
```

- Déclencher une exception (ou lever une exception) :

```
raise <Nom de l'exception>
```

Exceptions

- Rattraper une exception

```
1  try <expression 0> with
2  | <Nom de l'exception 1> -> <expression 1>
3  | <Nom de l'exception 2> -> <expression 2>
4  ...
```

Cette expression s'évalue en :

- `<expression 0>` si aucune exception n'est déclenchée
- `<expression 1>` si l'exception 1 est déclenchée
- `<expression 2>` si l'exception 2 est déclenchée
- ...

Exceptions

- Rattraper une exception

```
1  try <expression 0> with
2  | <Nom de l'exception 1> -> <expression 1>
3  | <Nom de l'exception 2> -> <expression 2>
4  ...
```

Cette expression s'évalue en :

- `<expression 0>` si aucune exception n'est déclenchée
- `<expression 1>` si l'exception 1 est déclenchée
- `<expression 2>` si l'exception 2 est déclenchée
- ...

Exercice

À l'aide d'une exception, écrire une fonction `mem: 'a -> 'a array -> bool` qui indique si un élément appartient à un tableau (fonction `Array.mem` en OCaml).

Solution de l'exercice

```
1  exception Appartient;;
2
3  let aux x tab =
4    for i = 0 to Array.length tab - 1 do
5      if tab.(i) = x then raise Appartient
6    done;
7    false;;
8
9  let mem x tab =
10   try aux x tab with
11   | Appartient -> true;;
```

```
1  # mem 0 [|1;2;3|];;
2  - : bool = false
```

```
1  # mem 1 [|1;2;3|];;
2  - : bool = true
```

Exceptions

- Une exception peut prendre un argument

```
1  exception <Nom de l'exception> of <type>
```

Exercice

À l'aide d'une exception, écrire une fonction `index: 'a -> 'a array -> int` qui prend en entrée `x` et `tab` et qui renvoie :

- L'indice de `x` dans `tab` s'il existe
- La taille de `tab` si `x` n'appartient pas à `tab`

Solution de l'exercice

```
exception Indice of int;;

let aux x tab =
  let n = Array.length tab in
  for i = 0 to n-1 do
    if tab.(i) = x then raise (Indice i);
  done;
  n;;

let index x tab =
  try aux x tab with
  | Indice i -> i;;
```

```
# index 0 [|1;2;3|];;
- : int = 3
```

```
# index 2 [|1;2;3|];;
- : int = 1
```

Exceptions

- Le `failwith` est défini de la manière suivante :

```
1  exception Failure of string;;
2  let failwith s = raise (Failure s);;
```

Exceptions

- Le `failwith` est défini de la manière suivante :

```
1  exception Failure of string;;
2  let failwith s = raise (Failure s);;
```

- Si possible, évitez les exceptions

Exercice

Sans utiliser d'exception, écrire une fonction `index: 'a -> 'a array -> int` qui prend en entrée `x` et `tab` et qui renvoie :

- L'indice de `x` dans `tab` s'il existe
- La taille de `tab` si `x` n'appartient pas à `tab`

Solution de l'exercice

```
1  let index x tab =
2    let i = ref 0 in
3    while !i < Array.length tab && tab.(!i) <> x do
4      incr i
5    done;
6    !i;;
```