Introduction à la programmation fonctionnelle

Notes de cours

Cours 3

28 février 2020

Sylvain Conchon

sylvain.conchon@lri.fr

Représentation de données structurées

- les types de base (`int`, `float`, etc.) sont insuffisants pour représenter des données structurées (dates, matrices etc.)
- des codages existent, mais ils ne sont pas naturels (et parfois aussi très inefficaces)

dans ce cours nous allons étudier trois structures de données (et leurs opérations associées)

1. les produits (*n-uplets*)
2. les produits nommés (*enregistrements*)
3. les sommes (*constructeurs*)

Les Types Produits

Le produit cartésien

La structure de données la plus simple pour former des valeurs complexes est la paire (ou *produit cartésien*)

Le produit cartésien

La structure de données la plus simple pour former des valeurs complexes est la paire (ou *produit cartésien*)

```
# (1, 2) ;;
```

Le produit cartésien

La structure de données la plus simple pour former des valeurs complexes est la paire (ou *produit cartésien*)

```
# (1, 2) ;;
```

```
 - : int * int = (1,2)
```

Le produit cartésien

La structure de données la plus simple pour former des valeurs complexes est la paire (ou *produit cartésien*)

```
# (1, 2) ;;

 - : int * int = (1,2)
```

Les composantes des paires peuvent être de types différents

Le produit cartésien

La structure de données la plus simple pour former des valeurs complexes est la paire (ou *produit cartésien*)

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

Les composantes des paires peuvent être de types différents

```
# ('a', 2.7) ;;
```

Le produit cartésien

La structure de données la plus simple pour former des valeurs complexes est la paire (ou *produit cartésien*)

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

Les composantes des paires peuvent être de types différents

```
# ('a', 2.7) ;;
- : char * float = ('a',2.7)
```

Accéder aux composantes d’une paire

Les fonctions `fst` et `snd` permettent respectivement d’accéder à la première et la deuxième composante d’une paire

Accéder aux composantes d’une paire

Les fonctions `fst` et `snd` permettent respectivement d’accéder à la première et la deuxième composante d’une paire

```
# snd (1,2);;
```

Accéder aux composantes d'une paire

Les fonctions `fst` et `snd` permettent respectivement d'accéder à la première et la deuxième composante d'une paire

```
# snd (1,2);;

 - : int = 2
```

Accéder aux composantes d'une paire

Les fonctions `fst` et `snd` permettent respectivement d'accéder à la première et la deuxième composante d'une paire

```
# snd (1,2);;

 - : int = 2


# fst ((1,2.5),'a');;
```

Accéder aux composantes d'une paire

Les fonctions `fst` et `snd` permettent respectivement d'accéder à la première et la deuxième composante d'une paire

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

```
# fst ((1,2.5),'a');;
 - : int * float = (1,2.5)
```

Accéder aux composantes d'une paire

Les fonctions `fst` et `snd` permettent respectivement d'accéder à la première et la deuxième composante d'une paire

```
# snd (1,2);;

 - : int = 2


# fst ((1,2.5),'a');;

 - : int * float = (1,2.5)


# let p = (1,2) in (snd p, fst p);;
```

Accéder aux composantes d'une paire

Les fonctions `fst` et `snd` permettent respectivement d'accéder à la première et la deuxième composante d'une paire

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

# fst ((1,2.5),'a');;
 - : int * float = (1,2.5)

# let p = (1,2) in (snd p, fst p);;
 - : int * int = (2,1)
```

n-uplets

Le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en n-uplets

n-uplets

Le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en n-uplets

```
# (2+3,false||true,"bonjour");;
```

n-uplets

Le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en n-uplets

```
# (2+3,false||true,"bonjour");;

 - : int * bool * string = (5,true,"bonjour")
```

Le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en n-uplets

```
# (2+3,false||true,"bonjour");;

 - : int * bool * string = (5,true,"bonjour")
```

Les n-uplets et les paires peuvent se mélanger

n-uplets

Le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en n-uplets

```
# (2+3,false||true,"bonjour");;
 - : int * bool * string = (5,true,"bonjour")
```

Les n-uplets et les paires peuvent se mélanger

```
# ('a',1.2,(true,0));;
```

n-uplets

Le produit cartésien (binaire) se généralise facilement afin de regrouper les valeurs en n-uplets

```
# (2+3,false||true,"bonjour");;
 - : int * bool * string = (5,true,"bonjour")
```

Les n-uplets et les paires peuvent se mélanger

```
# ('a',1.2,(true,0));;
 - : char * float * (bool * int) = ('a',1.2,(true,0))
```

Ne pas confondre...

. . . les types suivants

`int * int * int`   `(int * int) * int`   `int * (int * int)`

Ne pas confondre...

. . . les types suivants

`int * int * int` `(int * int) * int` `int * (int * int)`

- le premier désigne un triplet d'entiers

Ne pas confondre...

…les types suivants

`int * int * int`   `(int * int) * int`   `int * (int * int)`

- le premier désigne un triplet d'entiers
- le second est une paire dont la **première** composante est une **paire** d'entiers et la deuxième un entier

Ne pas confondre...

. . . les types suivants

`int * int * int` `(int * int) * int` `int * (int * int)`

- le premier désigne un triplet d'entiers
- le second est une paire dont la **première** composante est une **paire** d'entiers et la deuxième un entier
- le troisième est une paire dont la première composante est un entier et la **deuxième** composante est une **paire** d'entiers

Accès aux éléments d'un n-uplet (1/2)

On utilise une forme généralisée du `let`

```
let motif = e
```

ou une construction `match-with`

```
match e with
  motif -> ...
```

où le *motif* permet de filtrer le n-uplet représenté par l'expression `e`

Accès aux éléments d’un n-uplet (1/2)

On utilise une forme généralisée du `let`

`let` *motif* `= e`

ou une construction `match-with`

```
match e with
  motif -> ...
```

où le *motif* permet de filtrer le n-uplet représenté par l’expression `e`

```
# let v = ('a',1.2,"bonjour");;
val v : char * float * string = ('a',1.2,"bonjour")
```

Accès aux éléments d'un n-uplet (1/2)

On utilise une forme généralisée du `let`

`let` *motif* `= e`

ou une construction `match-with`

```
match e with
  motif -> ...
```

où le *motif* permet de filtrer le n-uplet représenté par l'expression `e`

```
# let v = ('a',1.2,"bonjour");;
val v : char * float * string = ('a',1.2,"bonjour")
```

On récupère les éléments de `v` avec `(x,y,z)` comme motif

```
# let (x,y,z) = v;;
val x : char = 'a'
val y : float = 1.2
val z : string = "bonjour"
```

Accès aux éléments d'un n-uplet (2/2)

```
# let v = (1,('a',2.3));;
val v : int * (char * float) = (1,('a',2.3))
```

Accès aux éléments d’un n-uplet (2/2)

```
# let v = (1,('a',2.3));;
val v : int * (char * float) = (1,('a',2.3))
```

Accès aux composantes d’une paire

Accès aux éléments d’un n-uplet (2/2)

```
# let v = (1,('a',2.3));;
val v : int * (char * float) = (1,('a',2.3))
```

Accès aux composantes d’une paire

```
# let (x,c) = v;;
val x : int = 1
val c : char * float = ('a',2.3)
```

Accès aux éléments d'un n-uplet (2/2)

```
# let v = (1,('a',2.3));;
val v : int * (char * float) = (1,('a',2.3))
```

Accès aux composantes d'une paire

```
# let (x,c) = v;;
val x : int = 1
val c : char * float = ('a',2.3)
```

Accès aux composantes d'une composante

Accès aux éléments d'un n-uplet (2/2)

```
# let v = (1,('a',2.3));;
val v : int * (char * float) = (1,('a',2.3))
```

Accès aux composantes d'une paire

```
# let (x,c) = v;;
val x : int = 1
val c : char * float = ('a',2.3)
```

Accès aux composantes d'une composante

```
# let (x,(y,z)) = v;;
val x : int = 1
val y : char = 'a'
val z : float = 2.3
```

N-uplets : valeurs de première classe

Les n-uplets peuvent être passés en arguments aux fonctions

N-uplets : valeurs de première classe

Les n-uplets peuvent être passés en arguments aux fonctions

```
# let f (x,y,z) = x + y * (int_of_float z);;

val f : int * int * float -> int = <fun>
```

N-uplets : valeurs de première classe

Les n-uplets peuvent être passés en arguments aux fonctions

```
# let f (x,y,z) = x + y * (int_of_float z);;

val f : int * int * float -> int = <fun>

# f (1,2,3.5);;
 - : int = 7
```

ou retournés comme résultats

N-uplets : valeurs de première classe

Les n-uplets peuvent être passés en arguments aux fonctions

```
# let f (x,y,z) = x + y * (int_of_float z);;

val f : int * int * float -> int = <fun>

# f (1,2,3.5);;
 - : int = 7
```

ou retournés comme résultats

```
# let rec division n m =
   if n<m then (0,n)
   else
    let (q,r) = division (n - m) m in (q + 1,r);;

val division : int -> int -> int * int = <fun>
```

Exemple

Calcul efficace de la fonction de Fibonacci dans `fibonacci.ml`

```
let fibonacci n =
 let rec fib_rapide n =
  if n=0 then (0,1)
  else let (x,y) = fib_rapide (n-1) in (y,x+y)
 in
 fst (fib_rapide n)

print_int (fibonacci 15);;
```

Exemple

Calcul efficace de la fonction de Fibonacci dans `fibonacci.ml`

```
let fibonacci n =
 let rec fib_rapide n =
  if n=0 then (0,1)
  else let (x,y) = fib_rapide (n-1) in (y,x+y)
 in
 fst (fib_rapide n)

print_int (fibonacci 15);;
```

compilation

```
> ocamlc -o fibonacci fibonacci.ml
```

Exemple

Calcul efficace de la fonction de Fibonacci dans `fibonacci.ml`

```
let fibonacci n =
 let rec fib_rapide n =
  if n=0 then (0,1)
  else let (x,y) = fib_rapide (n-1) in (y,x+y)
 in
 fst (fib_rapide n)

print_int (fibonacci 15);;
```

compilation

```
> ocamlc -o fibonacci fibonacci.ml
```

exécution

```
> ./fibonacci
610
```

Inconvénients des n-uplets (1/2)

- Les objets représentés par des n-uplets ne sont pas identifiés de manière unique
- Le système de types du langage ne peut donc pas être utilisé pour garantir de ”bonnes” propriétés

Exemple : on représente les nombres complexes par des paires `(r,i)` de type `float*float` où `r` et `i` sont respectivement la partie réelle et la partie imaginaire du complexe

- Malheureusement ce type peut tout aussi bien représenter des nombres complexes en notation polaire, ou des intervalles etc.
- Comment alors garantir que la fonction suivante est bien utilisée pour ajouter des complexes ?

```
# let add (r1,i1) (r2,i2) = (r1+.r2 , i1+.i2);;
val add: float*float -> float*float -> float*float = <fun>
```

Inconvénients des n-uplets (2/2)

Les n-uplets avec un grand nombre de composantes deviennent très vite inutilisables en pratique

Exemple : une fiche d'un fichier de personnes (nom, prénom, adresse, date de naissance, téléphone fixe, téléphone portable, etc.)

```
# let v =
    ("Durand","Jacques",
       ("2 rue J.Monod", "Orsay Cedex", 91893),
       (10,03,1967), "0130452637","0645362738" ...)
```

- La consultation des informations devient vite pénible
- Plusieurs éléments du n-uplet peuvent avoir le même type (ex. nom et prénom) et il est facile de les confondre (sans que le système de type puisse nous aider)

Produits Nommés

les enregistrements

Le produit nommé permet de définir des enregistrements : des n-uplets dont les éléments (**champs**) ont chacun un nom *distinct*

En OCaml, chaque produit nommé (ou **type enregistrement**) possède un nom donné par l'utilisateur

les enregistrements

Le produit nommé permet de définir des enregistrements : des n-uplets dont les éléments (**champs**) ont chacun un nom *distinct*

En OCaml, chaque produit nommé (ou **type enregistrement**) possède un nom donné par l'utilisateur

```
# type complexe = { re : float; im : float};;
type complexe = { re : float; im : float}
```

les enregistrements

Le produit nommé permet de définir des enregistrements : des n-uplets dont les éléments (**champs**) ont chacun un nom *distinct*

En OCaml, chaque produit nommé (ou **type enregistrement**) possède un nom donné par l'utilisateur

```
# type complexe = { re : float; im : float};;
type complexe = { re : float; im : float}
```

On crée des valeurs de type `complexe` de la manière suivante

les enregistrements

Le produit nommé permet de définir des enregistrements : des n-uplets dont les éléments (**champs**) ont chacun un nom *distinct*

En OCAML, chaque produit nommé (ou **type enregistrement**) possède un nom donné par l'utilisateur

```
# type complexe = { re : float; im : float};;
type complexe = { re : float; im : float}
```

On crée des valeurs de type `complexe` de la manière suivante

```
# { re = 1.4; im = 0.5};;
 - : complexe = { re = 1.4; im = 0.5}
```

Accès aux éléments d'un enregistrement (1/2)

L'accès le plus simple aux champs d'un enregistrement se fait à l'aide de la notation

objet • nom_du_champ

Accès aux éléments d'un enregistrement (1/2)

L'accès le plus simple aux champs d'un enregistrement se fait à l'aide de la notation

objet **.** *nom_du_champ*

```
# let v = { re = 1.3; im = 0.9};;
val v : complexe = { re = 1.3; im = 0.9}
```

Accès aux éléments d'un enregistrement (1/2)

L'accès le plus simple aux champs d'un enregistrement se fait à l'aide de la notation

objet **.** *nom_du_champ*

```
# let v = { re = 1.3; im = 0.9};;
val v : complexe = { re = 1.3; im = 0.9}
```

```
# v.re;;
- : float = 1.3
```

Accès aux éléments d'un enregistrement (2/2)

Le filtrage permet un accès partiel et en profondeur aux champs d'un enregistrement

```
# type t = { a : int; b : float * char; c : string };;
type t = { a : int; b : float * char; c : string }
```

Accès aux éléments d'un enregistrement (2/2)

Le filtrage permet un accès partiel et en profondeur aux champs d'un enregistrement

```
# type t = { a : int; b : float * char; c : string };;
type t = { a : int; b : float * char; c : string }

# let v = { a = 1; b = (3.4,'a'); c = "bonjour" };;
val v : t = { a = 1; b = (3.4,'a'); c = "bonjour" }
```

Accès aux éléments d'un enregistrement (2/2)

Le filtrage permet un accès partiel et en profondeur aux champs d'un enregistrement

```
# type t = { a : int; b : float * char; c : string };;
type t = { a : int; b : float * char; c : string }

# let v = { a = 1; b = (3.4,'a'); c = "bonjour" };;
val v : t = { a = 1; b = (3.4,'a'); c = "bonjour" }

# let { b = (_,x); c = y} = v;;
val x : char = 'a'
val y : string = "bonjour"
```

L'ordre des champs n'a pas d'importance

Les définitions de types suivantes sont équivalentes

```
type t = { a : int; b : char; c : bool }
```

```
type t = { b : char; c : bool; a : int }
```

L’ordre des champs n’a pas d’importance

Les définitions de types suivantes sont équivalentes

```
type t = { a : int; b : char; c : bool }
```

```
type t = { b : char; c : bool; a : int }
```

De même ces valeurs sont égales :

```
# { a = 1; b = 't'; c = true} = { b = 't'; c = true; a = 1} ;;
- : bool = true
```

L’ordre des champs n’a pas d’importance

Les définitions de types suivantes sont équivalentes

```
type t = { a : int; b : char; c : bool }
```

```
type t = { b : char; c : bool; a : int }
```

De même ces valeurs sont égales :

```
# { a = 1; b = 't'; c = true} = { b = 't'; c = true; a = 1} ;;
- : bool = true
```

Le filtrage est également insensible à l’ordre des champs :

```
# let { c = x; b = y} = { b = 't'; c = true; a = 1} ;;
val x : bool = true
val y : char = 't'
```

Le mélange des n-uplets et des enregistrements permet de définir des objets complexes

```
# type adresse = { rue : string; ville : string; cp : int};;
# type fiche = {
    nom : string;
    prenom : string ;
    adresse : adresse;
    date_naissance : int * int * int;
    tel_fixe : string;
    portable : string
  };;
```

Création de nouvelles valeurs

```
# let v1 = { a = 1; b = false; c = 'r'};;
val v1 : t = { a = 1; b = false; c = 'r'}
```

Création de nouvelles valeurs

```
# let v1 = { a = 1; b = false; c = 'r'};;
val v1 : t = { a = 1; b = false; c = 'r'}
```

On peut créer un nouvel enregistrement `v2` en utilisant le contenu des champs de `v1`

Création de nouvelles valeurs

```
# let v1 = { a = 1; b = false; c = 'r'};;
val v1 : t = { a = 1; b = false; c = 'r'}
```

On peut créer un nouvel enregistrement `v2` en utilisant le contenu des champs de `v1`

```
# let v2 = { a = v1.a; b = true; c = v1.c};;
val v2 : t = { a = 1; b = true; c = 'r'}
```

Création de nouvelles valeurs

```
# let v1 = { a = 1; b = false; c = 'r'};;
val v1 : t = { a = 1; b = false; c = 'r'}
```

On peut créer un nouvel enregistrement `v2` en utilisant le contenu des champs de `v1`

```
# let v2 = { a = v1.a; b = true; c = v1.c};;
val v2 : t = { a = 1; b = true; c = 'r'}
```

Le raccourci syntaxique suivant permet d'arriver au même résultat

```
{v with c1 = e1; ... cn=en}
```

Création de nouvelles valeurs

```
# let v1 = { a = 1; b = false; c = 'r'};;
val v1 : t = { a = 1; b = false; c = 'r'}
```

On peut créer un nouvel enregistrement `v2` en utilisant le contenu des champs de `v1`

```
# let v2 = { a = v1.a; b = true; c = v1.c};;
val v2 : t = { a = 1; b = true; c = 'r'}
```

Le raccourci syntaxique suivant permet d'arriver au même résultat

```
{v with c1 = e1; ... cn=en}
```

```
# let v3 = { v1 with b = true };;
val v3 : t = { a = 1; b = true; c = 'r'}
```

Enregistrements : valeurs de première classe

Une fonction prenant un enregistrement en argument :

```
# let f v = v.a;;
val f : t -> int
```

Enregistrements : valeurs de première classe

Une fonction prenant un enregistrement en argument :

```
# let f v = v.a;;
val f : t -> int

# f {a = 1; b = false; c = 'e'};;
- : int = 1
```

Enregistrements : valeurs de première classe

Une fonction prenant un enregistrement en argument :

```
# let f v = v.a;;
val f : t -> int

# f {a = 1; b = false; c = 'e'};;
- : int = 1
```

Les enregistrements peuvent aussi être retournés en résultat

```
# let f {a=x} v = { v with a = x+v.a } ;;
val f : t -> t -> t
```

Enregistrements : valeurs de première classe

Une fonction prenant un enregistrement en argument :

```
# let f v = v.a;;
val f : t -> int

# f {a = 1; b = false; c = 'e'};;
- : int = 1
```

Les enregistrements peuvent aussi être retournés en résultat

```
# let f {a=x} v = { v with a = x+v.a } ;;
val f : t -> t -> t

# let v = {a=1;b=true;c='r'} in f v v;;
- : t = {a=2;b=true;c='r'}
```

Sommes

Les types sommes

- Modélisation de domaines finis
- Réalisation de sommes disjointes permettant de réunir dans un même type des valeurs pouvant appartenir à des types différents

Constructeurs constants

On peut modéliser un domaine fini comportant exactement `n` valeurs avec un type somme

Exemple, les couleurs d'un jeu de carte :

```
# type couleur = Pique | Coeur | Carreau | Trefle;;
type couleur = Pique | Coeur | Carreau | Trefle
```

- les identificateurs `Pique`, `Coeur`, `Carreau` et `Trefle` sont des constructeurs (les majuscules sont obligatoires pour définir des constructeurs)
- le nom du domaine fini est `couleur`

Utilisation des constructeurs

L'unique manière de créer des valeurs d'un type somme est d'utiliser un constructeur :

Utilisation des constructeurs

L'unique manière de créer des valeurs d'un type somme est d'utiliser un constructeur :

```
# Trefle;;
- : couleur = Trefle
```

Utilisation des constructeurs

L'unique manière de créer des valeurs d'un type somme est d'utiliser un constructeur :

```
# Trefle;;
- : couleur = Trefle
```

```
# let v = (Pique , Coeur);;
val v : couleur * couleur = (Pique , Coeur)
```

Utilisation des constructeurs

L'unique manière de créer des valeurs d'un type somme est d'utiliser un constructeur :

```
# Trefle;;
- : couleur = Trefle

# let v = (Pique , Coeur);;
val v : couleur * couleur = (Pique , Coeur)

# Pique = Coeur;;
- : bool = false
```

Filtrage des constructeurs constants

La construction match-with permet de définir de manière compacte une **analyse par cas** d'un type somme

Filtrage des constructeurs constants

La construction match-with permet de définir de manière compacte une **analyse par cas** d'un type somme

```
# let points v =
   match v with
     Pique -> 1
    | Trefle -> 2
    | Coeur -> 3
    | Carreau -> 4;;
```

Filtrage des constructeurs constants

La construction match-with permet de définir de manière compacte une **analyse par cas** d'un type somme

```
# let points v =
   match v with
     Pique -> 1
   | Trefle -> 2
   | Coeur -> 3
   | Carreau -> 4;;

val points : couleur -> int = <fun>
```

Filtrage des constructeurs constants

La construction match-with permet de définir de manière compacte une **analyse par cas** d'un type somme

```
# let points v =
   match v with
     Pique -> 1
   | Trefle -> 2
   | Coeur -> 3
   | Carreau -> 4;;

val points : couleur -> int = <fun>

# points Coeur;;
- : int = 3
```

Constructeurs avec arguments

Les constructeurs peuvent également avoir des arguments, par exemple :

```
# type num = Int of int | Float of float
type num = Int of int | Float of float
```

Le mot-clé `of` indique que le constructeur attend un argument

On crée des valeurs en appliquant les constructeurs à des arguments du bon type :

```
# Int(5);;
```

Constructeurs avec arguments

Les constructeurs peuvent également avoir des arguments, par exemple :

```
# type num = Int of int | Float of float
type num = Int of int | Float of float
```

Le mot-clé `of` indique que le constructeur attend un argument

On crée des valeurs en appliquant les constructeurs à des arguments du bon type :

```
# Int(5);;
```

```
- : num = Int(5)
```

Filtrage des constructeurs avec arguments (1/2)

On utilise la construction match-with pour récupérer les arguments associés à un constructeur

```
# match Int(5) with Int(x) -> x+2;;
- : int = 7
```

En réalité, la réponse que l'on obtient est celle-là :

Filtrage des constructeurs avec arguments (1/2)

On utilise la construction match-with pour récupérer les arguments associés à un constructeur

```
# match Int(5) with Int(x) -> x+2;;
- : int = 7
```

En réalité, la réponse que l'on obtient est celle-là :

```
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Float _
- : int = 7
```

Filtrage des constructeurs avec arguments (1/2)

On utilise la construction match-with pour récupérer les arguments associés à un constructeur

```
# match Int(5) with Int(x) -> x+2;;
- : int = 7
```

En réalité, la réponse que l'on obtient est celle-là :

```
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Float _
- : int = 7
```

Le filtrage exige de faire une analyse par cas complète en fonction du type de l'objet filtré et non de sa valeur.

Filtrage des constructeurs avec arguments (2/2)

La complétude de l'analyse peut être obtenue en exécutant `failwith "explication"` pour les cas impossibles

```
# match Int(5) with
    Int(x) -> x+2
  | Float(x) -> failwith "cas impossible";;
```

Filtrage des constructeurs avec arguments (2/2)

La complétude de l'analyse peut être obtenue en exécutant `failwith "explication"` pour les cas impossibles

```
# match Int(5) with
    Int(x) -> x+2
  | Float(x) -> failwith "cas impossible";;

- : int = 7
```

Exemple

L'addition de valeurs de type `num` peut s'écrire ainsi

Exemple

L'addition de valeurs de type `num` peut s'écrire ainsi

```
# let ajoute x y =
   match (x,y) with
     (Int(m) , Int(n)) -> Int(m + n)
   | (Int(m) , Float(n)) -> Float((float_of_int m) +. n)
   | (Float(m) , Int(n)) -> Float(m +. (float_of_int n))
   | (Float(m) , Float(n)) -> Float(m +. n);;
```

Exemple

L'addition de valeurs de type `num` peut s'écrire ainsi

```
# let ajoute x y =
   match (x,y) with
     (Int(m) , Int(n)) -> Int(m + n)
   | (Int(m) , Float(n)) -> Float((float_of_int m) +. n)
   | (Float(m) , Int(n)) -> Float(m +. (float_of_int n))
   | (Float(m) , Float(n)) -> Float(m +. n);;

val ajoute : num -> num -> num = <fun>
```

On utilise simplement cette fonction de la manière suivante

Exemple

L’addition de valeurs de type `num` peut s’écrire ainsi

```
# let ajoute x y =
   match (x,y) with
     (Int(m) , Int(n)) -> Int(m + n)
   | (Int(m) , Float(n)) -> Float((float_of_int m) +. n)
   | (Float(m) , Int(n)) -> Float(m +. (float_of_int n))
   | (Float(m) , Float(n)) -> Float(m +. n);;

val ajoute : num -> num -> num = <fun>
```

On utilise simplement cette fonction de la manière suivante

```
# ajoute (Float(3.5)) (Int(5));;
- : num = Float(8.5)
```

Autre exemple

Le type des cartes d'un jeu de cartes :

```
type valeur = Roi | Reine | Valet | Num of int
type couleur = Coeur | Pique | Trefle | Carreau
type carte = valeur * couleur

# let compare (c1,_) (c2,_) =
   if c1 = c2 then 0
   else  match c1,c2 with
     | Roi, _ -> 1
     | Reine, Roi -> -1
     | Reine, _ -> 1
     | Valet, Roi -> -1
     | Valet, Reine -> -1
     | Valet, _ -> 1
     | Num(x), Num(y) -> (x - y) / (abs (x - y))
     | _ -> -1
```

Le type des séquences (listes)

Séquences d'entiers

On peut utiliser un type somme pour représenter des séquences (non bornées) d'entiers :

```
# type int_list = Nil | Cons of int * int_list;;
```

- `Nil` représente la séquence vide
- `Cons(x,l)` est la séquence dont le premier élément (on dit aussi la tête) est `x` et la suite est la séquence `l`

Par exemple, la séquence `4;1;5;8;1` est représentée par la valeur :

```
# Cons(4,Cons(1,Cons(5,Cons(8,Cons(1,Nil)))));;
- : int_list = Cons(4,Cons(1,Cons(5,Cons(8,Cons(1,Nil)))))
```

Le type `int list`

Le type des séquences est prédéfini en OCaml et ses éléments se notent avec une syntaxe spéciale

- `Cons` se note `::` et est infixe
- `Nil` se note `[]`

Par exemple, la séquence `4;1;5;8;1` est représentée par :

```
# 4::1::5::8::1::[];;
- : int list = [4;1;5;8;1]
```

On peut aussi directement utiliser la notation `[e1;e2;...;en]`

```
# [4;1;5;8;1];;
- : int list = [4;1;5;8;1]
```

ou faire un mélange des deux notations :

```
# 4::1::[5;8;1];;
- : int list = [4;1;5;8;1]
```

Des listes de types quelconques

OCaml permet de définir des listes dont les éléments peuvent être autre chose que des entiers :

```
# ['c';'a';'m';'l'];;
- : char list = ['c';'a';'m';'l']

# [["des";"listes"];["de";"listes"]];;
- : string list list = [["des";"listes"];["de";"listes"]]
```

Mais il n'est pas possible de construire une liste d'éléments de types différents :

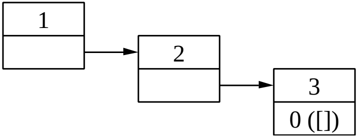
```
# [10;'a';4];;
      ---
This expression has type char but is here used
with type int
```

Listes chaînées

Les listes prédéfinies en OCAML correspondent exactement aux listes chaînées définies habituellement en C par le type suivant

```
typedef struct list{
  int elt;
  struct list* suivant;
};
```

La représentation mémoire de ces listes correspond à un chaînage de blocs mémoire, par exemple, la liste `[1; 2; 3]` correspond à :



Accès aux éléments d'une liste

On accède aux éléments d'une liste à l'aide des fonctions prédéfinies `List.hd` et `List.tl`

```
# List.hd [3;6;1;2];;
- : int = 3

# List.tl [3;6;1;2];;
- : int list = [6;1;2];;
```

`List.hd` et `List.tl` échouent sur une liste vide

```
# List.hd [];;
Exception: Failure "hd".

# List.tail [];;
Exception: Failure "tl".
```

Fonctions sur les listes

Définitions de fonctions sur les listes

La définition des fonctions sur les listes prennent généralement la forme d'une définition à deux cas :

- le cas où liste est vide
- le cas où elle ne l'est pas

Pour cette raison, il est plus agréable de réaliser cette analyse par cas avec du filtrage :

```
# let f l =
   match l with
     [] -> ...
   | x::s -> ...
```

la fonction zeros

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>
```

la fonction zeros

La fonction zeros vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie true si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>

# zeros [];;
- : bool = true
```

la fonction zeros

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>

# zeros [];;
- : bool = true

# zeros [0;0;0];;
- : bool = true
```

la fonction `zeros`

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>

# zeros [];;
- : bool = true

# zeros [0;0;0];;
- : bool = true

# zeros [0;1;0];;
- : bool = false
```

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

```
zeros [0;0;0]
```

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$\begin{array}{rcl} & & \texttt{zeros [0;0;0]} \\ [0;0;0] \neq [\,], \mathrm{x} = 0, \mathrm{s} = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\ & = & \texttt{zeros [0;0]} \end{array}$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [\,], \mathrm{x} = 0, \mathrm{s} = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [\,], \mathrm{x} = 0, \mathrm{s} = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]}
\end{array}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [], x = 0, s = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [], x = 0, s = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]} \\
[0] \neq [], x = 0, s = [] & \Rightarrow & \texttt{0=0 \&\& zeros []} \\
 & = & \texttt{zeros []}
\end{array}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [\,], x = 0, s = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [\,], x = 0, s = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]} \\
[0] \neq [\,], x = 0, s = [\,] & \Rightarrow & \texttt{0=0 \&\& zeros []} \\
 & = & \texttt{zeros []} \\
[\,] = [\,] & \Rightarrow & \texttt{true}
\end{array}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [], x = 0, s = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [], x = 0, s = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]} \\
[0] \neq [], x = 0, s = [] & \Rightarrow & \texttt{0=0 \&\& zeros []} \\
 & = & \texttt{zeros []} \\
[] = [] & \Rightarrow & \texttt{true}
\end{array}
$$

Évaluation de `zeros [0;1;0]`

$$
\texttt{zeros [0;1;0]}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [], x = 0, s = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [], x = 0, s = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]} \\
[0] \neq [], x = 0, s = [] & \Rightarrow & \texttt{0=0 \&\& zeros []} \\
 & = & \texttt{zeros []} \\
[] = [] & \Rightarrow & \texttt{true}
\end{array}
$$

Évaluation de `zeros [0;1;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;1;0]} \\
[0;1;0] \neq [], x = 0, s = [1;0] & \Rightarrow & \texttt{0=0 \&\& zeros [1;0]} \\
 & = & \texttt{zeros [1;0]}
\end{array}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{lcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [], \mathrm{x} = 0, \mathrm{s} = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [], \mathrm{x} = 0, \mathrm{s} = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]} \\
[0] \neq [], \mathrm{x} = 0, \mathrm{s} = [] & \Rightarrow & \texttt{0=0 \&\& zeros []} \\
 & = & \texttt{zeros []} \\
[] = [] & \Rightarrow & \texttt{true}
\end{array}
$$

Évaluation de `zeros [0;1;0]`

$$
\begin{array}{lcl}
 & & \texttt{zeros [0;1;0]} \\
[0;1;0] \neq [], \mathrm{x} = 0, \mathrm{s} = [1;0] & \Rightarrow & \texttt{0=0 \&\& zeros [1;0]} \\
 & = & \texttt{zeros [1;0]} \\
[1;0] \neq [], \mathrm{x} = 1, \mathrm{s} = [0] & \Rightarrow & \texttt{1=0 \&\& zeros [0]} \\
 & = & \texttt{false}
\end{array}
$$

Recherche d'un entier dans une liste

La fonction `recherche` détermine si un entier n figure bien dans une liste `l`

```
# let rec recherche n l =
   match l with
     [] -> false
   | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>
```

Recherche d’un entier dans une liste

La fonction `recherche` détermine si un entier n figure bien dans une liste `l`

```
# let rec recherche n l =
   match l with
     [] -> false
   | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>
```

```
# recherche 4 [3;2;4;1];;
- : bool = true
```

Recherche d'un entier dans une liste

La fonction `recherche` détermine si un entier n figure bien dans une liste `l`

```
# let rec recherche n l =
   match l with
     [] -> false
   | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>

# recherche 4 [3;2;4;1];;
- : bool = true

# recherche 4 [1;2];;
- : bool = false
```

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

```
recherche 4 [3;2;4;1]
```

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathrm{x} = 3, \mathrm{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathrm{x} = 3, \mathrm{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathrm{x} = 2, \mathrm{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

$$
\begin{array}{lcl}
 & & \texttt{recherche 4 [3;2;4;1]} \\
[3;2;4;1] \neq [], \mathrm{x} = 3, \mathrm{s} = [2;4;1] & \Rightarrow & \texttt{3=4 || recherche 4 [2;4;1]} \\
 & = & \texttt{recherche 4 [2;4;1]} \\
[2;4;1] \neq [], \mathrm{x} = 2, \mathrm{s} = [4;1] & \Rightarrow & \texttt{2=4 || recherche [4;1]} \\
 & = & \texttt{recherche 4 [4;1]} \\
[4;1] \neq [], \mathrm{x} = 4, \mathrm{s} = [1] & \Rightarrow & \texttt{4=4 || recherche 4 [1]} \\
 & = & \texttt{true}
\end{array}
$$

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

$$
\begin{array}{lcl}
 & & \texttt{recherche 4 [3;2;4;1]} \\
[3;2;4;1] \neq [], x = 3, s = [2;4;1] & \Rightarrow & \texttt{3=4 || recherche 4 [2;4;1]} \\
 & = & \texttt{recherche 4 [2;4;1]} \\
[2;4;1] \neq [], x = 2, s = [4;1] & \Rightarrow & \texttt{2=4 || recherche [4;1]} \\
 & = & \texttt{recherche 4 [4;1]} \\
[4;1] \neq [], x = 4, s = [1] & \Rightarrow & \texttt{4=4 || recherche 4 [1]} \\
 & = & \texttt{true}
\end{array}
$$

Évaluation de `recherche 4 [1;2]`

$$\texttt{recherche 4 [1;2]}$$

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], x = 3, s = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], x = 2, s = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`
$[4;1] \neq [], x = 4, s = [1]$	$\Rightarrow$	`4=4 || recherche 4 [1]`
	$=$	`true`

Évaluation de `recherche 4 [1;2]`

		`recherche 4 [1;2]`
$[1;2] \neq [], x = 1, s = [2]$	$\Rightarrow$	`1=4 || recherche 4 [2]`
	$=$	`recherche 4 [2]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

$$
\begin{array}{rcl}
 & & \texttt{recherche 4 [3;2;4;1]} \\
[3;2;4;1] \neq [], x = 3, s = [2;4;1] & \Rightarrow & \texttt{3=4 || } \textcolor{red}{\texttt{recherche 4 [2;4;1]}} \\
 & = & \textcolor{red}{\texttt{recherche 4 [2;4;1]}} \\
[2;4;1] \neq [], x = 2, s = [4;1] & \Rightarrow & \texttt{2=4 || } \textcolor{red}{\texttt{recherche [4;1]}} \\
 & = & \textcolor{red}{\texttt{recherche 4 [4;1]}} \\
[4;1] \neq [], x = 4, s = [1] & \Rightarrow & \texttt{4=4 || } \textcolor{red}{\texttt{recherche 4 [1]}} \\
 & = & \texttt{true}
\end{array}
$$

Évaluation de `recherche 4 [1;2]`

$$
\begin{array}{rcl}
 & & \texttt{recherche 4 [1;2]} \\
[1;2] \neq [], x = 1, s = [2] & \Rightarrow & \texttt{1=4 || } \textcolor{red}{\texttt{recherche 4 [2]}} \\
 & = & \textcolor{red}{\texttt{recherche 4 [2]}} \\
[2] \neq [], x = 2, s = [] & \Rightarrow & \texttt{2=4 || } \textcolor{red}{\texttt{recherche 4 []}} \\
 & = & \textcolor{red}{\texttt{recherche 4 []}}
\end{array}
$$

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

$$
\begin{array}{lcl}
 & & \texttt{recherche 4 [3;2;4;1]} \\
[3;2;4;1] \neq [], \mathrm{x} = 3, \mathrm{s} = [2;4;1] & \Rightarrow & \texttt{3=4 || recherche 4 [2;4;1]} \\
 & = & \texttt{recherche 4 [2;4;1]} \\
[2;4;1] \neq [], \mathrm{x} = 2, \mathrm{s} = [4;1] & \Rightarrow & \texttt{2=4 || recherche [4;1]} \\
 & = & \texttt{recherche 4 [4;1]} \\
[4;1] \neq [], \mathrm{x} = 4, \mathrm{s} = [1] & \Rightarrow & \texttt{4=4 || recherche 4 [1]} \\
 & = & \texttt{true}
\end{array}
$$

Évaluation de `recherche 4 [1;2]`

$$
\begin{array}{lcl}
 & & \texttt{recherche 4 [1;2]} \\
[1;2] \neq [], \mathrm{x} = 1, \mathrm{s} = [2] & \Rightarrow & \texttt{1=4 || recherche 4 [2]} \\
 & = & \texttt{recherche 4 [2]} \\
[2] \neq [], \mathrm{x} = 2, \mathrm{s} = [] & \Rightarrow & \texttt{2=4 || recherche 4 []} \\
 & = & \texttt{recherche 4 []} \\
[] = [] & \Rightarrow & \texttt{false}
\end{array}
$$

Longueur d’une liste

La fonction `longueur` retourne la longueur d’une liste

```
# let rec longueur l =
   match l with
     [] -> 0
   | _::s -> 1 + (longueur s);;
```

Longueur d'une liste

La fonction `longueur` retourne la longueur d'une liste

```
# let rec longueur l =
   match l with
     [] -> 0
   | _::s -> 1 + (longueur s);;
```

Une version récursive terminale :

```
# let longueur l =
   let rec longrec acc l =
     match l with
      [] -> acc
    | _::s -> longrec (1+acc) s
   in
   longrec 0 l
```

Longueur d'une liste

La fonction `longueur` retourne la longueur d'une liste

```
# let rec longueur l =
   match l with
     [] -> 0
   | _::s -> 1 + (longueur s);;
```

Une version récursive terminale :

```
# let longueur l =
   let rec longrec acc l =
     match l with
      [] -> acc
    | _::s -> longrec (1+acc) s
   in
   longrec 0 l
```

Cette fonction est prédéfinie en OCaml : `List.length`

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

```
   longueur [10;2;4]
=  longrec 0 [10;2;4]
```

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

$$
\begin{array}{rcl}
 & & \texttt{longueur [10;2;4]} \\
 & = & \texttt{longrec 0 [10;2;4]} \\
[10;2;4] \neq [], \mathrm{s} = [2;4] & \Rightarrow & \texttt{longrec (1+0) [2;4]}
\end{array}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

$$\begin{array}{rcl} & & \texttt{longueur [10;2;4]} \\ & = & \texttt{longrec 0 [10;2;4]} \\ [10;2;4] \neq [], s = [2;4] & \Rightarrow & \texttt{longrec (1+0) [2;4]} \\ [2;4] \neq [], s = [4] & \Rightarrow & \texttt{longrec (1+1) [4]} \end{array}$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

$$
\begin{array}{rcl}
 & & \texttt{longueur [10;2;4]} \\
 & = & \texttt{longrec 0 [10;2;4]} \\
[10;2;4] \neq [], \mathrm{s} = [2;4] & \Rightarrow & \texttt{longrec (1+0) [2;4]} \\
[2;4] \neq [], \mathrm{s} = [4] & \Rightarrow & \texttt{longrec (1+1) [4]} \\
[4] \neq [], \mathrm{s} = [] & \Rightarrow & \texttt{longrec (1+2) []}
\end{array}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

$$
\begin{array}{lcl}
 & & \texttt{longueur [10;2;4]} \\
 & = & \texttt{longrec 0 [10;2;4]} \\
[10;2;4] \neq [], s = [2;4] & \Rightarrow & \texttt{longrec (1+0) [2;4]} \\
[2;4] \neq [], s = [4] & \Rightarrow & \texttt{longrec (1+1) [4]} \\
[4] \neq [], s = [] & \Rightarrow & \texttt{longrec (1+2) []} \\
[] = [] & \Rightarrow & 3
\end{array}
$$

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

…alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple :

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple :

```
# longueur [[4.5;0.3;9.8];[];[3.2;1.8]];;
- : int = 3
```

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple :

```
# longueur [[4.5;0.3;9.8];[];[3.2;1.8]];;
- : int = 3
```

. . . dans ce cas la fonction `longueur` devrait également avoir

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

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

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

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

'a (qui se lit “apostrophe a”, ou encore “alpha”) est une variable de type

Une variable de type veut dire n'importe quel type

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCAML est le plus général :

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

'a (qui se lit “apostrophe a”, ou encore “alpha”) est une variable de type

Une variable de type veut dire n'importe quel type

Il faut donc lire le type de la fonction `longueur` comme suit :

“La fonction `longueur` prend en argument une **liste** – dont les éléments sont de n'importe quel type – et retourne un **entier**”

Fonctions génériques sur les listes

concaténation de listes

La fonction `append` construit une nouvelle la liste en réunissant deux listes bout à bout

```
# let rec append l1 l2 =
   match l1 with
     [] -> l2
   | x::s -> x::(append s l2);;
val append : 'a list -> 'a list -> 'a list = <fun>
```

concaténation de listes

La fonction `append` construit une nouvelle la liste en réunissant deux listes bout à bout

```
# let rec append l1 l2 =
   match l1 with
     [] -> l2
   | x::s -> x::(append s l2);;
val append : 'a list -> 'a list -> 'a list = <fun>

# append [2;5;1] [10;6;8;15];;
- : int list = [2;5;1;10;6;8;15]
```

- Cette fonction est prédéfinie en VERBATIM, il s'agit de `List.append`
- L'opérateur infixe `@` est un raccourci syntaxique pour cette fonction, on note `l1@l2` la concaténation de `l1` et `l2`

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

```
append [1;2] [3;4]
```

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

$$\begin{array}{rcl} & & \texttt{append [1;2] [3;4]} \\ [1;2] \neq [\,], \mathrm{x} = 1, \mathrm{s} = [2] & \Rightarrow & \texttt{1::(append [2] [3;4])} \end{array}$$

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

$$\begin{array}{rcl} & & \texttt{append [1;2] [3;4]} \\ [1;2] \neq [], x = 1, s = [2] & \Rightarrow & \texttt{1::(append [2] [3;4])} \\ [2] \neq [], x = 2, s = [] & \Rightarrow & \texttt{1::2::(append [] [3;4])} \end{array}$$

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

$$
\begin{array}{lll}
 & & \texttt{append [1;2] [3;4]} \\
[1;2] \neq [], \mathrm{x} = 1, \mathrm{s} = [2] & \Rightarrow & \texttt{1::(append [2] [3;4])} \\
[2] \neq [], \mathrm{x} = 2, \mathrm{s} = [] & \Rightarrow & \texttt{1::2::(append [] [3;4])} \\
[] = [] & \Rightarrow & \texttt{1::2::[3;4]} \\
 & \Rightarrow & \texttt{1::[2;3;4]} \\
 & \Rightarrow & \texttt{[1;2;3;4]}
\end{array}
$$

Concaténation rapide

- La fonction `append` n'est pas récursive terminale
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

Concaténation rapide

- La fonction `append` n'est pas récursive terminale
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | x :: s -> rev_append s (x :: l2)
val rev_append : 'a list -> 'a list -> 'a list = <fun>
```

Concaténation rapide

- La fonction `append` n'est pas récursive terminale
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | x :: s -> rev_append s (x :: l2)
val rev_append : 'a list -> 'a list -> 'a list = <fun>

# rev_append [4;2;6] [1;10;9;5];;
- : int list = [6; 2; 4; 1; 10; 9; 5]
```

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

```
# let rev l = rev_append l [];;
val rev : 'a list -> 'a list = <fun>
```

```
# rev [4;2;6;1];;
- : int list = [1; 6; 2; 4]
```

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$\begin{aligned} & \texttt{rev [1;2;3]} \\ = \;& \texttt{rev\_append [1;2;3] []} \end{aligned}$$

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$\begin{array}{rcl} & & \texttt{rev [1;2;3]} \\ & = & \texttt{rev\_append [1;2;3] []} \\ [1;2;3] \neq [], \mathrm{x} = 1, \mathrm{s} = [2;3] & \Rightarrow & \texttt{rev\_append [2;3] (1::[])} \\ & = & \texttt{rev\_append [2;3] [1]} \end{array}$$

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$
\begin{array}{rcl}
 & & \texttt{rev [1;2;3]} \\
 & = & \texttt{rev\_append [1;2;3] []} \\
{[1;2;3]} \neq [], \mathrm{x} = 1, \mathrm{s} = [2;3] & \Rightarrow & \texttt{rev\_append [2;3] (1::[])} \\
 & = & \texttt{rev\_append [2;3] [1]} \\
{[2;3]} \neq [], \mathrm{x} = 2, \mathrm{s} = [3] & \Rightarrow & \texttt{rev\_append [3] (2::[1])} \\
 & = & \texttt{rev\_append [3] [2;1]}
\end{array}
$$

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$
\begin{array}{rcl}
 & & \texttt{rev [1;2;3]} \\
 & = & \texttt{rev\_append [1;2;3] []} \\
[1;2;3] \neq [], x = 1, s = [2;3] & \Rightarrow & \texttt{rev\_append [2;3] (1::[])} \\
 & = & \texttt{rev\_append [2;3] [1]} \\
[2;3] \neq [], x = 2, s = [3] & \Rightarrow & \texttt{rev\_append [3] (2::[1])} \\
 & = & \texttt{rev\_append [3] [2;1]} \\
[3] \neq [], x = 3, s = [] & \Rightarrow & \texttt{rev\_append [] (3::[2;1])} \\
 & = & \texttt{rev\_append [] [3;2;1]}
\end{array}
$$

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$
\begin{array}{rcl}
 & & \texttt{rev [1;2;3]} \\
 & = & \texttt{rev\_append [1;2;3] []} \\
{[1;2;3]} \neq [\,], \mathrm{x} = 1, \mathrm{s} = [2;3] & \Rightarrow & \texttt{rev\_append [2;3] (1::[])} \\
 & = & \texttt{rev\_append [2;3] [1]} \\
{[2;3]} \neq [\,], \mathrm{x} = 2, \mathrm{s} = [3] & \Rightarrow & \texttt{rev\_append [3] (2::[1])} \\
 & = & \texttt{rev\_append [3] [2;1]} \\
{[3]} \neq [\,], \mathrm{x} = 3, \mathrm{s} = [\,] & \Rightarrow & \texttt{rev\_append [] (3::[2;1])} \\
 & = & \texttt{rev\_append [] [3;2;1]} \\
{[\,]} = [\,] & \Rightarrow & \texttt{[3;2;1]}
\end{array}
$$