Programmation Fonctionnelle
Cours 03

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

2 octobre 2014

Type produit ou n-uplets

Construire *n*-uplets

- Les produits cartésiens sont une parmi les constructions les plus simples de structures de données.

```
# let p = 1, "Hello";;
val p : int * string = (1, "Hello")
```

- On les construit en combinant les éléments par la virgule (souvent écrit entre parenthèses, mais pas nécessaire):

```
# 3.4, (fun x y -> x*y), 'a';;
- : float * (int -> int -> int) * char = (3.4, <fun>, 'a'
```

- Une valuer du type produit est une *n*-uplet de valuers des types correspondants:

```
# "Hello"^" world",(fun x ->x*x)3;;
- : string * int = ("Hello world", 9)
```

Construire *n*-uplets

- Les produits cartésiens sont une parmi les constructions les plus simples de structures de données.

```
# let p = 1, "Hello";;
val p : int * string = (1, "Hello")
```

- On les construit en combinant les éléments par la virgule (souvent écrit entre parenthèses, mais pas nécessaire):

```
# 3.4, (fun x y -> x*y), 'a';;
- : float * (int -> int -> int) * char = (3.4, <fun>, 'a'
```

- Une valuer du type produit est une *n*-uplet de valuers des types correspondants:

```
# "Hello"^" world",(fun x ->x*x)3;;
- : string * int = ("Hello world", 9)
```

Construire *n*-uplets

- Les produits cartésiens sont une parmi les constructions les plus simples de structures de données.

```
# let p = 1, "Hello";;
val p : int * string = (1, "Hello")
```

- On les construit en combinant les éléments par la virgule (souvent écrit entre parenthèses, mais pas nécessaire):

```
# 3.4, (fun x y -> x*y), 'a';;
- : float * (int -> int -> int) * char = (3.4, <fun>, 'a'
```

- Une valuer du type produit est une *n*-uplet de valuers des types correspondants:

```
# "Hello"^"␣world",(fun x ->x*x)3;;
- : string * int = ("Hello␣world", 9)
```

Un produit n'est pas une liste

- Attention: à ne pas confondre les produits avec les listes !

```
# let pair = 1,2;;
val pair : int * int = (1, 2)
# let liste = [1;2];;
val liste : int list = [1; 2]
# List.hd pair;;
          ^^^
Error: This expression has type int * int but
an expression was expected of type 'a list
```

- Les listes sont séquences *de longueur variable* et des éléments *du même type*

```
# 1,"Hello";;
- : int * string = (1, "Hello")
# [1;"Hello"];;
      ^^^^
Error: This expression has type string but an
expression was expected of type int
```

- Les *n*-uplets sont séquences dont la longueur est fixée par le type et les éléments peuvent être hétérogènes

Un produit n'est pas une liste

- Attention: à ne pas confondre les produits avec les listes !

```
# let pair = 1,2;;
val pair : int * int = (1, 2)
# let liste = [1;2];;
val liste : int list = [1; 2]
# List.hd pair;;
          ^^^
Error: This expression has type int * int but
an expression was expected of type 'a list
```

- Les listes sont séquences *de longueur variable* et des éléments *du même type*

```
# 1,"Hello";;
- : int * string = (1, "Hello")
# [1;"Hello"];;
       ^^^^
Error: This expression has type string but an
expression was expected of type int
```

- Les *n*-uplets sont séquences dont la longueur est fixée par le type et les éléments peuvent être hétérogènes

Un produit n'est pas une liste

- Attention: à ne pas confondre les produits avec les listes !

```
# let pair = 1,2;;
val pair : int * int = (1, 2)
# let liste = [1;2];;
val liste : int list = [1; 2]
# List.hd pair;;
          ^^^
Error: This expression has type int * int but
an expression was expected of type 'a list
```

- Les listes sont séquences *de longueur variable* et des éléments *du même type*

```
# 1,"Hello";;
- : int * string = (1, "Hello")
# [1;"Hello"];;
       ^^^^
Error: This expression has type string but an
expression was expected of type int
```

- Les *n*-uplets sont séquences dont la longueur est fixée par le type et les éléments peuvent être hétérogènes

Deconstruire *n*-uplets

Pour déconstruire les *n*-uplets, on peut

- utiliser les projections, dans le cas binaires:

```
# fst p;;
- : int = 1

# snd p;;
- : string = "Hello"
```

- ou, dans le cas général, le filtrage par motif:

```
# let trd triplet = match triplet with (x,y,z) -> z;;
val trd : 'a * 'b * 'c -> 'c = <fun>

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

- ou le let (raccourci pour le filtrage à un seul cas):

```
# let trd (x,y,z) = z;; (* raccourci *)
val trd : 'a * 'b * 'c -> 'c = <fun>
```

Deconstruire *n*-uplets

Pour déconstruire les *n*-uplets, on peut

- utiliser les projections, dans le cas binaires:

```
# fst p;;
- : int = 1

# snd p;;
- : string = "Hello"
```

- ou, dans le cas général, le filtrage par motif:

```
# let trd triplet = match triplet with (x,y,z) -> z;;
val trd : 'a * 'b * 'c -> 'c = <fun>

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

- ou le let (raccourci pour le filtrage à un seul cas):

```
# let trd (x,y,z) = z;; (* raccourci *)
val trd : 'a * 'b * 'c -> 'c = <fun>
```

Deconstruire *n*-uplets

Pour déconstruire les *n*-uplets, on peut

- utiliser les projections, dans le cas binaires:

```
# fst p;;
- : int = 1

# snd p;;
- : string = "Hello"
```

- ou, dans le cas général, le filtrage par motif:

```
# let trd triplet = match triplet with (x,y,z) -> z;;
val trd : 'a * 'b * 'c -> 'c = <fun>

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

- ou le let (raccourci pour le filtrage à un seul cas):

```
# let trd (x,y,z) = z;; (* raccourci *)
val trd : 'a * 'b * 'c -> 'c = <fun>
```

Ça sert à quoi les *n*-uplets ?

- Écrire des fonctions qui envoient *n*-uplets de valuers en sortie:

```
# let rec split pivot liste = match liste with
    | [] -> [],[]
  | h::r -> let (r1,r2) = split pivot r
        in if h<pivot then (h::r1, r2) else (r1, h::r2);;
val split : 'a -> 'a list -> 'a list * 'a list = <fun>

# split 17 [4;6;19; 0; 3049];;]
- : int list * int list = ([4; 6; 0], [19; 3049])
```

- Écrire fonctions qui prennent *n*-uplets de valeurs en argument (Attention : pas confondre avec fonctions plusieurs arguments)

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

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

- Construire des types plus complexes:
  - les listes d'association
  - les types algébriques

Ça sert à quoi les *n*-uplets ?

- Écrire des fonctions qui envoient *n*-uplets de valuers en sortie:

```
# let rec split pivot liste = match liste with
    | [] -> [],[]
  | h::r -> let (r1,r2) = split pivot r
        in if h<pivot then (h::r1, r2) else (r1, h::r2);;
val split : 'a -> 'a list -> 'a list * 'a list = <fun>

# split 17 [4;6;19; 0; 3049];;]
- : int list * int list = ([4; 6; 0], [19; 3049])
```

- Écrire fonctions qui prennent *n*-uplets de valeurs en argument (Attention : pas confondre avec fonctions plusieurs arguments)

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

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

- Construire des types plus complexes:
  - les listes d'association
  - les types algébriques

Ça sert à quoi les *n*-uplets ?

- Écrire des fonctions qui envoient *n*-uplets de valuers en sortie:

```
# let rec split pivot liste = match liste with
    | [] -> [],[]
  | h::r -> let (r1,r2) = split pivot r
        in if h<pivot then (h::r1, r2) else (r1, h::r2);;
val split : 'a -> 'a list -> 'a list * 'a list = <fun>

# split 17 [4;6;19; 0; 3049];;]
- : int list * int list = ([4; 6; 0], [19; 3049])
```

- Écrire fonctions qui prennent *n*-uplets de valeurs en argument (Attention : pas confondre avec fonctions plusieurs arguments)

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

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

- Construire des types plus complexes:
  - les listes d'association
  - les types algébriques

Type enregistrement

Enregistrements = produits avec champs nommés

```
type typename = {identifier1 : type1;
                 ... ;
                 identifiern : typen}
```

- en Anglais: record
- défini par la mot clé `type` comme un ensemble de champs nommés par des identificateurs
- on ne peut pas utiliser le même nom de champs dans deux types d'enregistrement différents
  - nécessaire pour une inférence de type efficace
- une valeur est un ensemble associant à chance champ une valeur du type correspondant
- accès aux champs (projections) en notation pointée ou par filtrage par motif

Enregistrements (Exemples)

```
# type date = {
    day: int;
    month: string;
    year: int
  };;
type date = { day : int; month : string; year : int; }

# let today = {
    day = 7; month = "october"; year = 2014;
  };;
val today : date = {day = 7; month = "october"; year = 2014}

# let tomorrow = {
  year = 2014;  day = 8;  month = "october";
  };;                              (*ordre des champs pas important *)
val tomorrow : date = {day = 8; month = "october"; year = 2014}

# today.year;;                                    (*notation pointee*)
- : int = 2014

# let getday {year=y; day=d; month=o} = d;;(*filtrage implicite*)
val getday : date -> int = <fun>
```

Enregistrements (Exemples)

```
# type r1 =
  {a: string; b:int};;
type r1 = { a : string; b : int; }

# type r2 =
  {a:string; b:float};;
type r2 = { a : string; b : float; }

# {a="john"; b=17;};;
                ^^^
Error: This expression has type int but an expression
         was expected of type float

# {a="john"; b=10.7};;
- : r2 = {a = "john"; b = 10.7}
```

Enregistrements vs n-uplets

Avantages des enregistrements:

- pas besoin de se rappeler l'ordre des éléments d'un enregistrement
- même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement
- moins de modifications à faire dans le code quand on ajoute un champ à un enregistrement, que quand on ajoute un composant à un n-uplet

Enregistrements vs *n*-uplets

Avantages des enregistrements:

- pas besoin de se rappeler l'ordre des éléments d'un enregistrement
- même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement
- moins de modifications à faire dans le code quand on ajoute un champ à un enregistrement, que quand on ajoute un composant à un *n*-uplet

Enregistrements vs *n*-uplets

Avantages des enregistrements:

- pas besoin de se rappeler l'ordre des éléments d'un enregistrement
- même pas besoin de connaître le nombre exact de champs pour accéder à un champ d'un enregistrement
- moins de modifications à faire dans le code quand on ajoute un champ à un enregistrement, que quand on ajoute un composant à un *n*-uplet

Enregistrements (Exercise)

1. Définir un type nombre complexe
2. Définir la fonction qui calcule la valeur absolue d'un nombre complexe
3. Définir la fonction qui calcule la multiplication de deux nombres complexes

Type somme et type algebrique

Type somme

```
type typename =
  | Identifier1 of type1
  | Identifier2 of type2
  ....
  | Identifiern of typen
```

- défini par un ensemble de cas séparés par une barre | en utilisant le mot clé `type`
  (première barre | optionnelle)
- chaque cas caractérisé par un constructeur
  l'identificateur doit commencer par une lettre majuscule
- chaque constructeur peut (pas obligatoire) avoir un argument d'un certain type.

Type somme

```
# type number =
    | Zero
    | Integer of int
    | Real of float;;
type number = Zero | Integer of int | Real of float
```

Les valeurs sont données par les constructeurs appliqués aux valeurs des arguments si besoin

```
# let x=Zero;;
val x : number = Zero

# let y = Real (3./.2.5);;
val y : number = Real 1.2
```

Les fonctions peuvent être définies par filtrage par motif

```
# let float_of_number = function
    | Zero -> 0.0
    | Integer x -> float_of_int x
    | Real x -> x;;
val float_of_number : number -> float = <fun>
```

Type somme

```
# type number =
    | Zero
    | Integer of int
    | Real of float;;
type number = Zero | Integer of int | Real of float
```

Les valeurs sont données par les constructeurs appliqués aux valeurs des arguments si besoin

```
# let x=Zero;;
val x : number = Zero

# let y = Real (3./.2.5);;
val y : number = Real 1.2
```

Les fonctions peuvent être définies par filtrage par motif

```
# let float_of_number = function
    | Zero -> 0.0
    | Integer x -> float_of_int x
    | Real x -> x;;
val float_of_number : number -> float = <fun>
```

Type somme

```
# type number =
    | Zero
    | Integer of int
    | Real of float;;
type number = Zero | Integer of int | Real of float
```

Les valeurs sont données par les constructeurs appliqués aux valeurs des arguments si besoin

```
# let x=Zero;;
val x : number = Zero

# let y = Real (3./.2.5);;
val y : number = Real 1.2
```

Les fonctions peuvent être définies par filtrage par motif

```
# let float_of_number = function
    | Zero -> 0.0
    | Integer x -> float_of_int x
    | Real x -> x;;
val float_of_number : number -> float = <fun>
```

Somme vs produit

- un type somme est une union disjointe des valeurs de ses composants, lorsque un type produit (ou un enregistrement) et un produit cartésien
- par exemple, supposons que type `t1` contient n_1 valeurs et `t2` contient n_2 valeurs, alors:
  - le type:

    ```
    type tp = {first t1; second t2};;
    ```

    contient $n_1 * n_2$ valeurs
  - le type:

    ```
    type ts = First of t1 | Second of t2;;
    ```

    contient $n_1 + n_2$ valeurs.

Somme vs produit

- un type somme est une union disjointe des valeurs de ses composants, lorsque un type produit (ou un enregistrement) et un produit cartésien
- par exemple, supposons que type `t1` contient n_1 valeurs et `t2` contient n_2 valeurs, alors:
  - le type:

    ```
    type tp = {first t1; second t2};;
    ```

    contient $n_1 * n_2$ valeurs
  - le type:

    ```
    type ts = First of t1 | Second of t2;;
    ```

    contient $n_1 + n_2$ valeurs.

Somme vs produit

- un type somme est une union disjointe des valeurs de ses composants, lorsque un type produit (ou un enregistrement) et un produit cartésien
- par exemple, supposons que type `t1` contient n_1 valeurs et `t2` contient n_2 valeurs, alors:
  - le type:

    ```
    type tp = {first t1; second t2};;
    ```

    contient $n_1 * n_2$ valeurs
  - le type:

    ```
    type ts = First of t1 | Second of t2;;
    ```

    contient $n_1 + n_2$ valeurs.

Type algébrique

La definition du type permet

des appels récursifs:

```
# type liste_int =
    | Vide
    | Cons of (int * liste_int);;
type liste_int = Vide | Cons of (int * liste_int)

# let l = Cons (42, Cons (-3, Vide));;
val l : liste_int = Cons (42, Cons (-3, Vide))
```

des variables de types:

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

On parlera dans ce cas d'un type algébrique.

Exercice: écrire les fonctions `map` et `fold_right` sur `'a liste`.

Type algébrique

La definition du type permet

des appels récursifs:

```
# type liste_int =
    | Vide
    | Cons of (int * liste_int);;
type liste_int = Vide | Cons of (int * liste_int)

# let l = Cons (42, Cons (-3, Vide));;
val l : liste_int = Cons (42, Cons (-3, Vide))
```

des variables de types:

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

On parlera dans ce cas d'un type algébrique.

Exercice: écrire les fonctions `map` et `fold_right` sur `'a liste`.

Un exemple: les arbres

1. Définir le type des arbres binaires étiquetés
2. Définir la fonction `taille` sur ce type
3. Définir la fonction `tas` prenant en entrée un arbre binaire étiqueté par des entiers et renvoyant `true` ou `false` selon que l'arbre est un tas ou pas.
   - On rappelle que un tas, en anglais *heap*, est un arbre binaire complet à gauche et qui vérifie la condition suivante: l'étiquette d'un nœud est supérieure ou égale à l'étiquette de chacun de ses fils.

Polymorphisme

Well-typed programs do not go wrong !

- OCaml c'est un langage fortement typé
  - toute expression a un type
  - les types sont synthétisés automatiquement
  - le système vérifie le respect du typage *avant* l'évaluation

- Catégories des types de OCaml:
  - Types de base (comme `int` ou `string`)
  - Paramètres (ou variables) de types (comme `'a`)
  - Types fonctionnels (comme `int -> int`, `('a -> 'b)-> 'b`)
  - Types algébriques (comme `char list`, `'a tree`)

- Les types donnent un contrôle sur les programmes:
  - `int*int` type d'une paire des entiers
  - `'a list->'a` type d'une fonction ayant comme argument une liste des elements d'un quelque type `'a` et rendant en sortie un élément de ce type `'a`

Well-typed programs do not go wrong !

- OCaml c'est un langage fortement typé
  - toute expression a un type
  - les types sont synthétisés automatiquement
  - le système vérifie le respect du typage *avant* l'évaluation
- Catégories des types de OCaml:
  - Types de base (comme `int` ou `string`)
  - Paramètres (ou variables) de types (comme `'a`)
  - Types fonctionnels (comme `int -> int`, `('a -> 'b)-> 'b`)
  - Types algébriques (comme `char list`, `'a tree`)
- Les types donnent un contrôle sur les programmes:
  - `int*int` type d'une paire des entiers
  - `'a list->'a` type d'une fonction ayant comme argument une liste des elements d'un quelque type `'a` et rendant en sortie un élément de ce type `'a`

Well-typed programs do not go wrong !

- OCaml c'est un langage fortement typé
  - toute expression a un type
  - les types sont synthétisés automatiquement
  - le système vérifie le respect du typage *avant* l'évaluation
- Catégories des types de OCaml:
  - Types de base (comme `int` ou `string`)
  - Paramètres (ou variables) de types (comme `'a`)
  - Types fonctionnels (comme `int -> int`, `('a -> 'b)-> 'b`)
  - Types algébriques (comme `char list`, `'a tree`)
- Les types donnent un contrôle sur les programmes:
  - `int*int` type d'une paire des entiers
  - `'a list->'a` type d'une fonction ayant comme argument une liste des elements d'un quelque type `'a` et rendant en sortie un élément de ce type `'a`

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

- la variable `y` est utilisée dans deux expressions entières (`y*y` et `y+1`), donc c'est de type `int`
- la variable `x` est utilisée comme conditionnel d'un `if`, donc c'est de type `bool`
- avec ces hypothèses le `if-then-else` est bien typé, son type c'est le type des deux branches: `int`
- le type de tout l'expression `fun` est

  `bool->int->int`

  c.à-d. le type d'une fonction qui prend un argument de type `bool` (variable `x`), un argument de type `int` (variable `y`) et renvoie un résultat de type `int` (le résultat du `if-then-else`).

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

- la variable `y` est utilisée dans deux expressions entières (`y*y` et `y+1`), donc c'est de type `int`
- la variable `x` est utilisée comme conditionnel d'un `if`, donc c'est de type `bool`
- avec ces hypothèses le `if-then-else` est bien typé, son type c'est le type des deux branches: `int`
- le type de tout l'expression `fun` est

  `bool->int->int`

  c.à-d. le type d'une fonction qui prend un argument de type `bool` (variable `x`), un argument de type `int` (variable `y`) et renvoie un résultat de type `int` (le résultat du `if-then-else`).

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

- la variable `y` est utilisée dans deux expressions entières (`y*y` et `y+1`), donc c'est de type `int`
- la variable `x` est utilisée comme conditionnel d'un `if`, donc c'est de type `bool`
- avec ces hypothèses le `if-then-else` est bien typé, son type c'est le type des deux branches: `int`
- le type de tout l'expression `fun` est

  `bool -> int -> int`

  c.à-d. le type d'une fonction qui prend un argument de type `bool` (variable `x`), un argument de type `int` (variable `y`) et renvoie un résultat de type `int` (le résultat du `if-then-else`).

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

- la variable `y` est utilisée dans deux expressions entières (`y*y` et `y+1`), donc c'est de type `int`
- la variable `x` est utilisée comme conditionnel d'un `if`, donc c'est de type `bool`
- avec ces hypothèses le `if-then-else` est bien typé, son type c'est le type des deux branches: `int`
- le type de tout l'expression `fun` est

  `bool->int->int`

  c.à-d. le type d'une fonction qui prend un argument de type `bool` (variable `x`), un argument de type `int` (variable `y`) et renvoie un résultat de type `int` (le résultat du `if-then-else`).

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

- la variable `y` est utilisée dans deux expressions entières (`y*y` et `y+1`), donc c'est de type `int`
- la variable `x` est utilisée comme conditionnel d'un `if`, donc c'est de type `bool`
- avec ces hypothèses le `if-then-else` est bien typé, son type c'est le type des deux branches: `int`
- le type de tout l'expression `fun` est

  `bool->int->int`

  c.à-d. le type d'une fonction qui prend un argument de type `bool` (variable `x`), un argument de type `int` (variable `y`) et renvoie un résultat de type `int` (le résultat du `if-then-else`).

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

- la variable `y` est utilisée dans deux expressions entières (`y*y` et `y+1`), donc c'est de type `int`
- la variable `x` est utilisée comme conditionnel d'un `if`, donc c'est de type `bool`
- avec ces hypothèses le `if-then-else` est bien typé, son type c'est le type des deux branches: `int`
- le type de tout l'expression `fun` est

  `bool->int->int`

  c.à-d. le type d'une fonction qui prend un argument de type `bool` (variable `x`), un argument de type `int` (variable `y`) et renvoie un résultat de type `int` (le résultat du `if-then-else`).

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

```
# fun x -> (x+1, x+.1.0);;
                  ^^^
Error: This expression has type int but an expression
was expected of type float
```

- dans la première composant de la paire, la variable `x` est utilisée dans une expression entiere (`x+1`), donc c'est de type `int`
- dans la seconde composant de la paire, on applique la variable `x` à un opérateur `float`,
- alert, le typeur echoue: un message d'erreur est envoyé

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

```
# fun x -> (x+1, x+.1.0);;
                  ^^^
Error: This expression has type int but an expression
was expected of type float
```

- dans la première composant de la paire, la variable `x` est utilisée dans une expression entiere (`x+1`), donc c'est de type `int`
- dans la seconde composant de la paire, on applique la variable `x` à un opérateur `float`,
- alert, le typeur echoue: un message d'erreur est envoyé

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x -> (x+1, x+.1.0);;
                  ^^^
Error: This expression has type int but an expression
was expected of type float
```

- dans la première composant de la paire, la variable `x` est utilisée dans une expression entiere (`x+1`), donc c'est de type `int`
- dans la seconde composant de la paire, on applique la variable `x` à un opérateur `float`,
- alert, le typeur echoue: un message d'erreur est envoyé

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x -> (x+1, x+.1.0);;
                  ^^^
Error: This expression has type int but an expression
was expected of type float
```

- dans la première composant de la paire, la variable `x` est utilisée dans une expression entiere (`x+1`), donc c'est de type `int`
- dans la seconde composant de la paire, on applique la variable `x` à un opérateur `float`,
- alert, le typeur echoue: un message d'erreur est envoyé

Synthèse de type d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

```
# fun x -> (x+1, x+.1.0);;
                  ^^^
Error: This expression has type int but an expression
was expected of type float
```

- dans la première composant de la paire, la variable `x` est utilisée dans une expression entiere (`x+1`), donc c'est de type `int`
- dans la seconde composant de la paire, on applique la variable `x` à un opérateur `float`,
- alert, le typeur echoue: un message d'erreur est envoyé

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a, 'b, ...`

Par exemple:

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

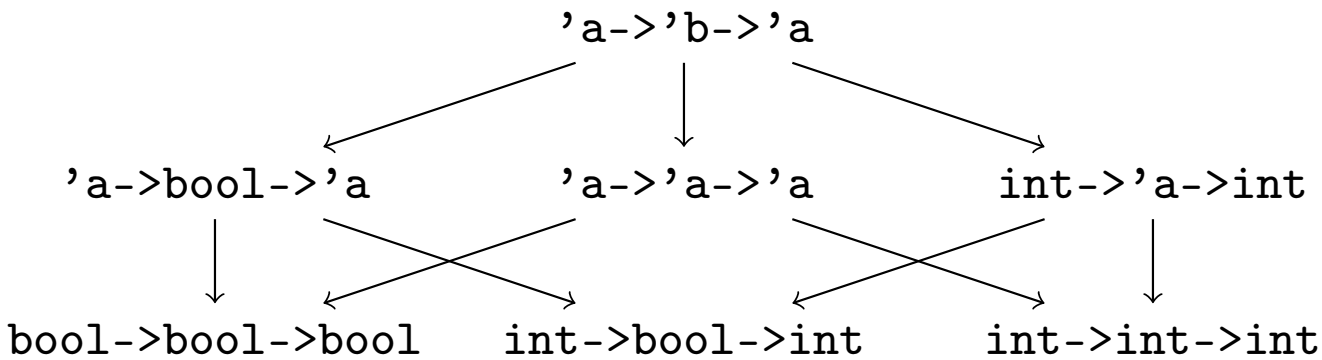
- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...

Par exemple:

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...

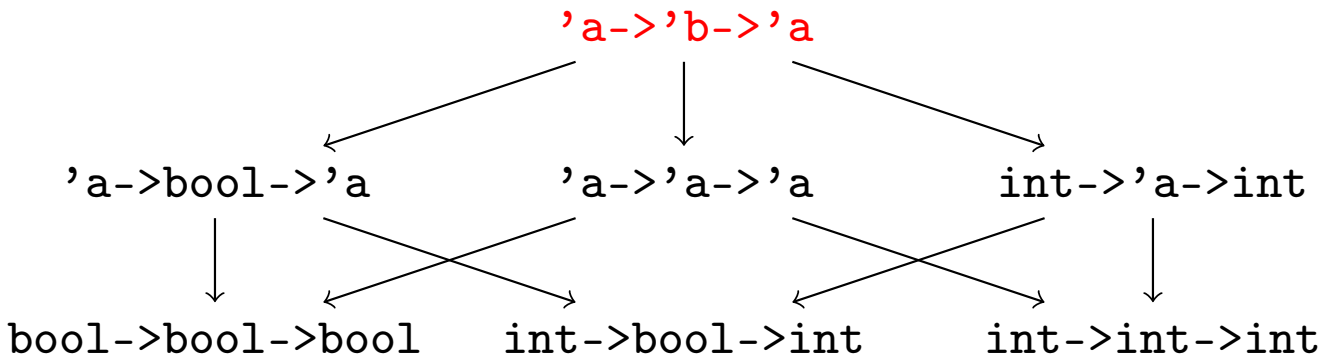


Par exemple:

Type principal d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



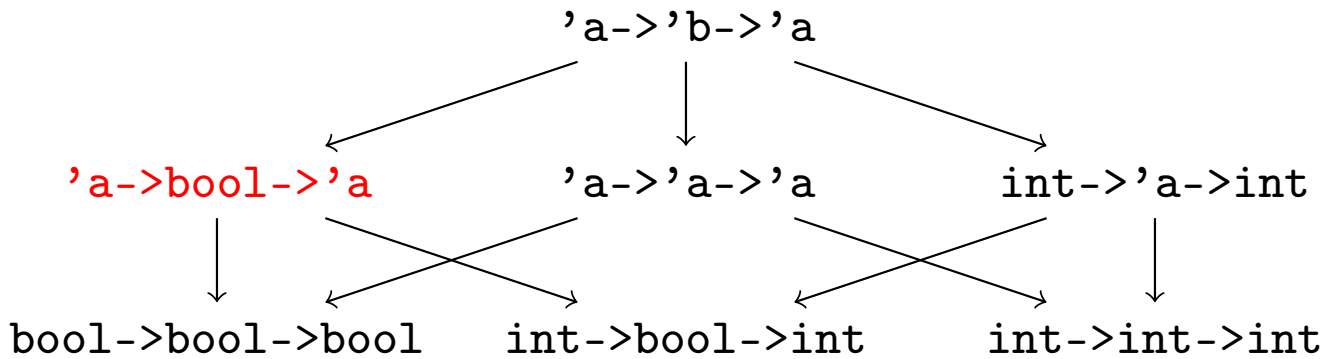
Par exemple:

```
# fun x y -> x;;
- : 'a -> 'b -> 'a = <fun>
```

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



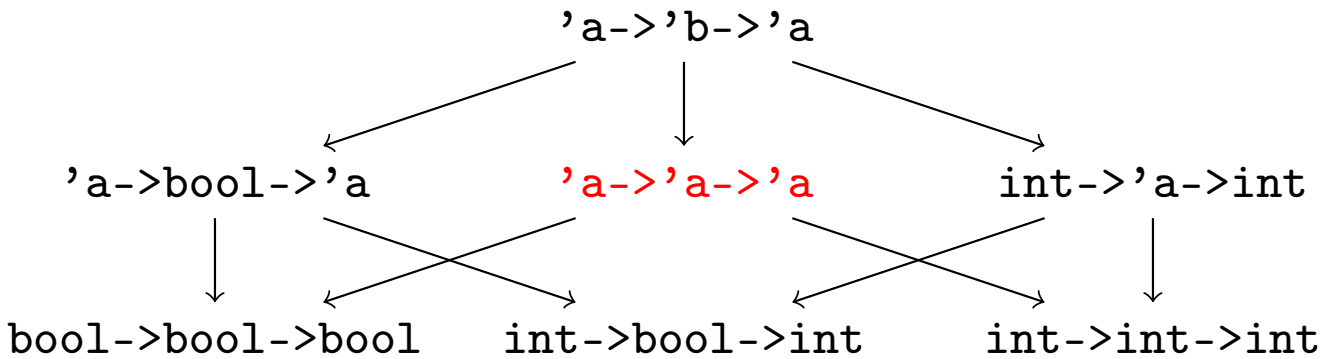
Par exemple:

```
# fun x y -> if y then x else x;;
- : 'a -> bool -> 'a = <fun>
```

Type principal d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



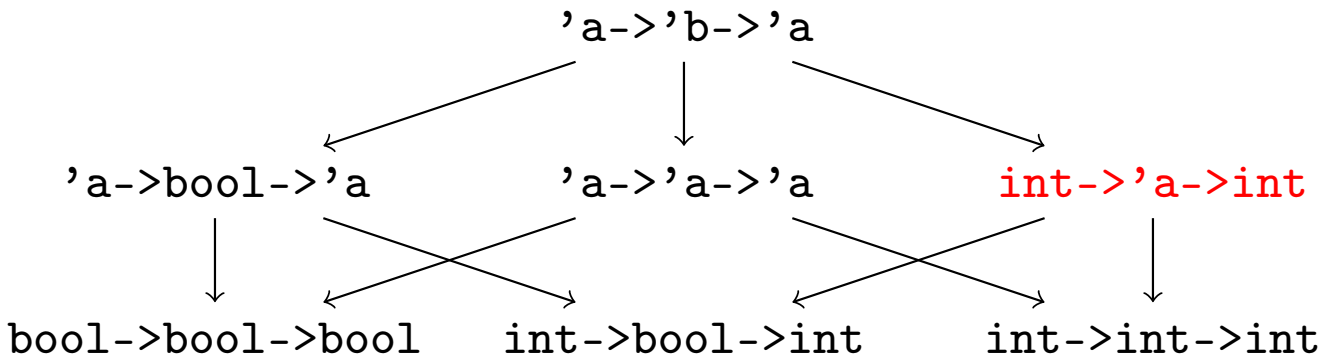
Par exemple:

```
# fun x y -> if x>y then x else y;;
- : 'a -> 'a -> 'a = <fun>
```

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



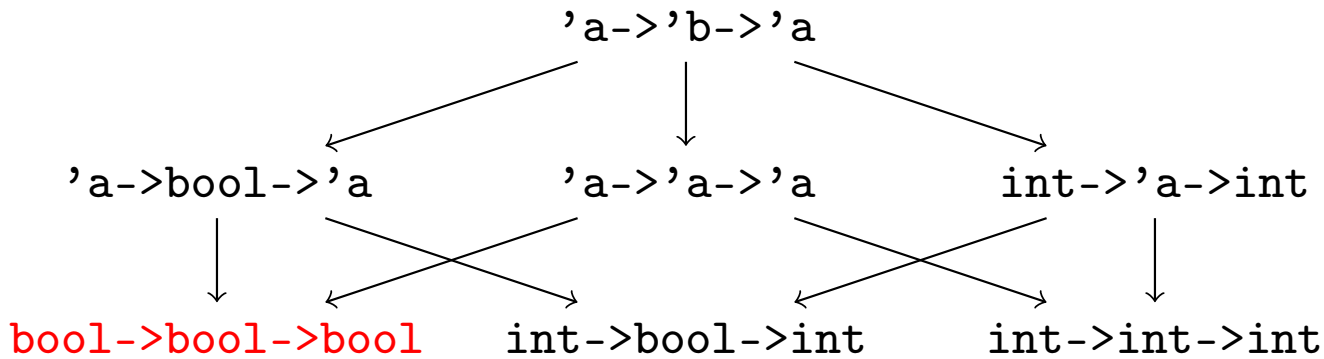
Par exemple:

```
# fun x y -> x+1;;
- : int -> 'a -> int = <fun>
```

Type principal d'une fonction

Expression `expr` $\longrightarrow$ Typeur $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



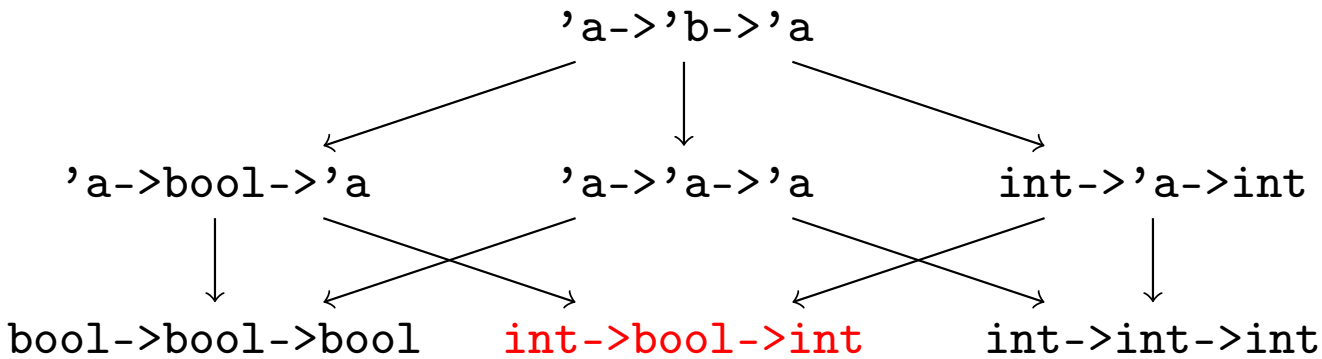
Par exemple:

```
# fun x y -> x && y;;
- : bool -> bool -> bool = <fun>
```

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



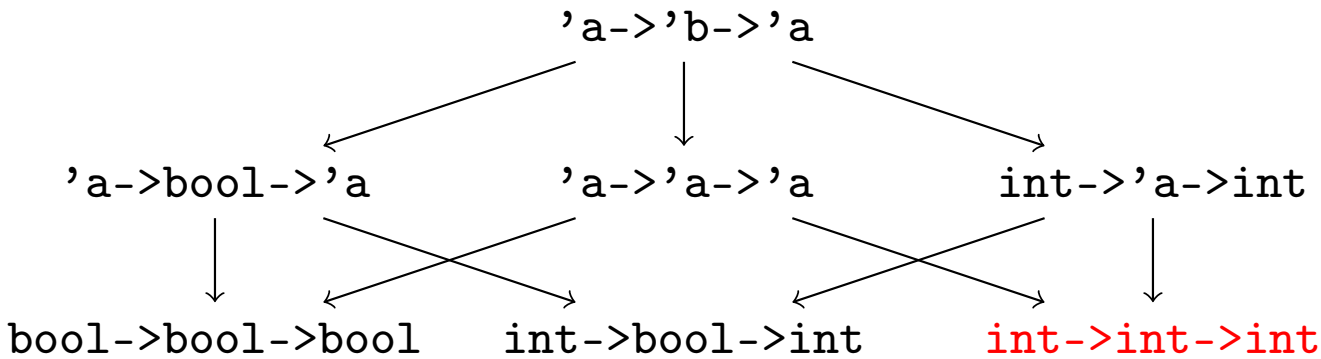
Par exemple:

```
# fun x y -> if y then x else x+1;;
- : int -> bool -> int = <fun>
```

Type principal d'une fonction

Expression `expr` $\longrightarrow$ **Typeur** $\longrightarrow$ un type τ pour `expr`

- le type τ engendré par le typeur doit être le plus général que tous les autres type possibles. Nous appelons ça un type principal.
- tous autres types sont obtenus par substitution des paramètres `'a`, `'b`, ...



Par exemple:

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

Polymorphisme paramétrique

- la substitution des variables de type permet le polymorphisme: une même fonction peut être appliquée à des entrées assez variées:

```
# let proj x y = x;;
val proj : 'a -> 'b -> 'a = <fun>
# proj 1 "toto";;
- : int = 1
# proj true 3.0;;
- : bool = true
# proj (fun x-> x+1) ['a';'c';'t'];;
- : int -> int = <fun>
```

- on appelle ce type de polymorphisme "paramétrique" car basé sur la substitution des variables de types (appelées aussi paramètres).
  - à ne pas confondre avec le soi-disant "polymorphisme ad-hoc" qui est la surcharge de noms (interdite en OCaml),
  - ni avec le "polymorphisme d'inclusion" induit par le sous-typage typique de la programmation orientée objet (permit en OCaml au niveau des classes...).

Polymorphisme paramétrique

- la substitution des variables de type permet le polymorphisme: une même fonction peut être appliquée à des entrées assez variées:

```
# let proj x y = x;;
val proj : 'a -> 'b -> 'a = <fun>
# proj 1 "toto";;
- : int = 1
# proj true 3.0;;
- : bool = true
# proj (fun x-> x+1) ['a';'c';'t'];;
- : int -> int = <fun>
```

- on appelle ce type de polymorphisme "paramétrique" car basé sur la substitution des variables de types (appelées aussi paramètres).
  - à ne pas confondre avec le soi-disant "polymorphisme ad-hoc" qui est la surcharge de noms (interdite en OCaml),
  - ni avec le "polymorphisme d'inclusion" induit par le sous-typage typique de la programmation orientée objet (permit en OCaml au niveau des classes...).

Polymorphisme paramétrique

- la substitution des variables de type permet le polymorphisme: une même fonction peut être appliquée à des entrées assez variées:

```
# let proj x y = x;;
val proj : 'a -> 'b -> 'a = <fun>
# proj 1 "toto";;
- : int = 1
# proj true 3.0;;
- : bool = true
# proj (fun x-> x+1) ['a';'c';'t'];;
- : int -> int = <fun>
```

- on appelle ce type de polymorphisme "paramétrique" car basé sur la substitution des variables de types (appelées aussi paramètres).
  - à ne pas confondre avec le soi-disant "polymorphisme ad-hoc" qui est la surcharge de noms (interdite en OCaml),
  - ni avec le "polymorphisme d'inclusion" induit par le sous-typage typique de la programmation orientée objet (permit en OCaml au niveau des classes...).

Polymorphisme paramétrique

- la substitution des variables de type permet le polymorphisme: une même fonction peut être appliquée à des entrées assez variées:

```
# let proj x y = x;;
val proj : 'a -> 'b -> 'a = <fun>
# proj 1 "toto";;
- : int = 1
# proj true 3.0;;
- : bool = true
# proj (fun x-> x+1) ['a';'c';'t'];;
- : int -> int = <fun>
```

- on appelle ce type de polymorphisme "paramétrique" car basé sur la substitution des variables de types (appelées aussi paramètres).
  - à ne pas confondre avec le soi-disant "polymorphisme ad-hoc" qui est la surcharge de noms (interdite en OCaml),
  - ni avec le "polymorphisme d'inclusion" induit par le sous-typage typique de la programmation orientée objet (permit en OCaml au niveau des classes...).

Exemples de fonctions polymorphes

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

# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>

# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# let pair x y = x,y;;
val pair : 'a -> 'b -> 'a * 'b = <fun>

# fst;;
- : 'a * 'b -> 'a = <fun>

# snd;;
- : 'a * 'b -> 'b = <fun>

# let geq x y = x>=y;;
val geq : 'a -> 'a -> bool = <fun>

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

Les limites du polymorphisme en OCaml

- ```
  # let id x =x;;
  val id : 'a -> 'a = <fun>
  # ((id true), (id 2));;
  - : bool * int = (true, 2)
  ```

  1. la première occurrence de `id` a le type `bool->bool`
  2. la seconde a le type `int->int`

- Cependant

  ```
  # let g f = ((f true),(f 2));;
                          ^^

  Error: This expression has type int but an expression
  was expected of type bool
  ```

- les contraints qui OCaml pose au polymorphisme empêchent que le type de la fonction `f` soit polymorphe
  - la synthèse des types de OCaml (basée sur l'unification) ne permet pas de partir de deux type particuliers et de trouver un type plus générale:
    1. `(f true)` force que soit de type `bool->'a`
    2. 2 force que `f` soit de type `int ->'a`
  ```

Les limites du polymorphisme en OCaml

- ```
  # let id x =x;;
  val id : 'a -> 'a = <fun>
  # ((id true), (id 2));;
  - : bool * int = (true, 2)
  ```

  1. la première occurrence de `id` a le type `bool->bool`
  2. la seconde a le type `int->int`

- Cependant

  ```
  # let g f = ((f true),(f 2));;
                           ^^
  Error: This expression has type int but an expression
  was expected of type bool
  ```

- les contraints qui OCaml pose au polymorphisme empêchent que le type de la fonction `f` soit polymorphe
  - la synthèse des types de OCaml (basée sur l'unification) ne permet pas de partir de deux type particuliers et de trouver un type plus générale:
    1. `(f true)` force que soit de type `bool->'a`
    2. 2 force que `f` soit de type `int ->'a`
  ```

Les limites du polymorphisme en OCaml

- ```
  # let id x =x;;
  val id : 'a -> 'a = <fun>
  # ((id true), (id 2));;
  - : bool * int = (true, 2)
  ```

  1. la première occurrence de `id` a le type `bool->bool`
  2. la seconde a le type `int->int`

- Cependant

  ```
  # let g f = ((f true),(f 2));;
                           ^^

  Error: This expression has type int but an expression
  was expected of type bool
  ```

- les contraints qui OCaml pose au polymorphisme empêchent que le type de la fonction `f` soit polymorphe
  - la synthèse des types de OCaml (basée sur l'unification) ne permet pas de partir de deux type particuliers et de trouver un type plus générale:
    1. `(f true)` force que soit de type `bool->'a`
    2. 2 force que `f` soit de type `int ->'a`
  ```

Annotation de type

- il est quand même permit de restreindre le type à travers des annotations de type explicites:

```
# let proj (x:int) y = x;;          (*annotation sur un argument*)
val proj : int -> 'a -> int = <fun>

# proj 3 "toto";;
- : int = 3

# proj "trois" "toto";;
           ^^^
Error: This expression has type string but an expression
was expected of type int

# let proj x y :int = x;;           (*annotation sur le resultat*)
val proj : int -> 'a -> int = <fun>
```

- si les annotations ne sont pas compatibles (entre eux ou avec le type principal) il y a un erreur:

```
# let succ x:float = x+1;;
                     ^^^
Error: This expression has type int but an expression
was expected of type float
```

Annotation de type

- il est quand même permit de restreindre le type à travers des annotations de type explicites:

```
# let proj (x:int) y = x;;          (*annotation sur un argument*)
val proj : int -> 'a -> int = <fun>

# proj 3 "toto";;
- : int = 3

# proj "trois" "toto";;
           ^^^
Error: This expression has type string but an expression
was expected of type int

# let proj x y :int = x;;           (*annotation sur le resultat*)
val proj : int -> 'a -> int = <fun>
```

- si les annotations ne sont pas compatibles (entre eux ou avec le type principal) il y a un erreur:

```
# let succ x:float = x+1;;
                     ^^^
Error: This expression has type int but an expression
was expected of type float
```

Annotation de type

- il est quand même permit de restreindre le type à travers des annotations de type explicites:

```
# let proj (x:int) y = x;;          (*annotation sur un argument*)
val proj : int -> 'a -> int = <fun>

# proj 3 "toto";;
- : int = 3

# proj "trois" "toto";;
           ^^^
Error: This expression has type string but an expression
was expected of type int

# let proj x y :int = x;;           (*annotation sur le resultat*)
val proj : int -> 'a -> int = <fun>
```

- si les annotations ne sont pas compatibles (entre eux ou avec le type principal) il y a un erreur:

```
# let succ x:float = x+1;;
                     ^^^
Error: This expression has type int but an expression
was expected of type float
```

Doggy bag

- types structurés
  - produits ou n-uplets
  - enregistrements
  - sommes
  - types algébriques
- polymoprhisme
  - synthèse de type
  - type principal d'une fonction
  - polymorphisme paramétrique
  - annotation de type

Exercice

1. Définir le type des arbres étiquetés avec arité arbitraire
2. Définir la fonction `taille` sur ce type
3. Définir la fonction `greffe` prenant en entrée deux arbres `t1` et `t2` et renvoyant en sortie l'arbre obtenu en ajoutant comme premier descendant de la racine de `t1` l'arbre `t2`.