

Programmation Fonctionnelle

Cours 10

Michele Pagani



Université Paris Diderot
UFR Informatique
IRIF

`pagani@irif`

5 décembre 2016

Récurrance terminale

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with  
  | 0 -> []  
  | n -> n::(mklist (n-1));;  
val mklist : int -> int list = <fun>
```

```
# mklist 3;;  
- : int list = [3; 2; 1]
```

```
# mklist 1000000;;  
Stack overflow during evaluation (looping recursion?).
```

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with  
| 0 -> []  
| n -> n::(mklist (n-1));;  
val mklist : int -> int list = <fun>
```

$\text{mklist } 3 \Rightarrow 3::(\text{mklist } 2) \Rightarrow 3::2::(\text{mklist } 1) \Rightarrow 3::2::1(\text{mklist } 0)$
 $\Rightarrow 3::2:: 1::[] \Rightarrow 3:: 2::[1] \Rightarrow 3::[2;1] \Rightarrow [3;2;1]$

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la **pile d'exécution** (angl. **call stack**)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriqués, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
| 0 -> []
| n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

$\text{mklist } 3 \Rightarrow \underline{3::}(\text{mklist } 2) \Rightarrow \underline{3::2::}(\text{mklist } 1) \Rightarrow \underline{3::2::1}(\text{mklist } 0)$
 $\Rightarrow \underline{3::2::} \underline{1::[]} \Rightarrow \underline{3::} \underline{2::[1]} \Rightarrow \underline{3::} [2;1] \Rightarrow [3;2;1]$

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la pile d'exécution (angl. call stack)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriqués, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

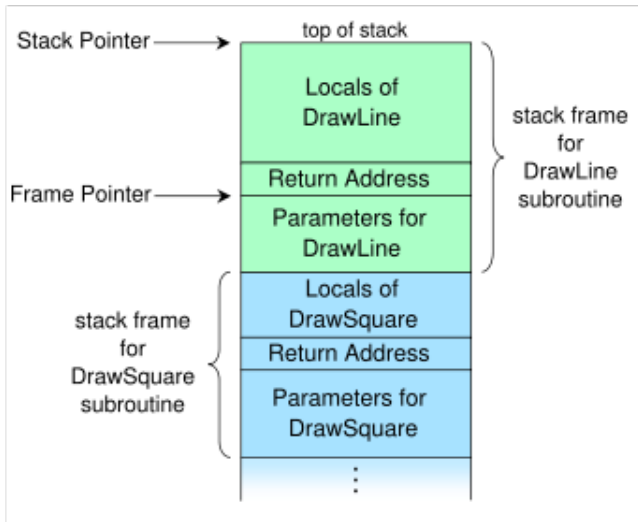
```
# let rec mklist n = match n with  
| 0 -> []  
| n -> n::(mklist (n-1));;  
val mklist : int -> int list = <fun>
```

$$\begin{aligned} \text{mklist } 3 &\Rightarrow \underline{3::}(\text{mklist } 2) \Rightarrow \underline{3::2::}(\text{mklist } 1) \Rightarrow \underline{3::2::1}(\text{mklist } 0) \\ &\Rightarrow \underline{3::2::} \underline{1::[]} \Rightarrow \underline{3::} \underline{2::[1]} \Rightarrow \underline{3::} [2;1] \Rightarrow [3;2;1] \end{aligned}$$

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la **pile d'exécution** (angl. **call stack**)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriqués, on peut épuiser l'espace disponible pour la pile !

Exemple: DrawSquare appelle DrawLine



Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
| 0 -> []
| n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

$$\begin{aligned} \text{mklist } 3 &\Rightarrow \underline{3::}(\text{mklist } 2) \Rightarrow \underline{3::2::}(\text{mklist } 1) \Rightarrow \underline{3::2::1}(\text{mklist } 0) \\ &\Rightarrow \underline{3::2::} \underline{1::[]} \Rightarrow \underline{3::} \underline{2::[1]} \Rightarrow \underline{3::} [2;1] \Rightarrow [3;2;1] \end{aligned}$$

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la **pile d'exécution** (angl. **call stack**)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Conséquence: si trop d'appels imbriqués, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
| 0 -> []
| n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

$\text{mklist } 3 \Rightarrow \underline{3::}(\text{mklist } 2) \Rightarrow \underline{3::2::}(\text{mklist } 1) \Rightarrow \underline{3::2::1}(\text{mklist } 0)$
 $\Rightarrow \underline{3::2::} \underline{1::[]} \Rightarrow \underline{3::} \underline{2::[1]} \Rightarrow \underline{3::} [2;1] \Rightarrow [3;2;1]$

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la **pile d'exécution** (angl. **call stack**)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Conséquence: si trop d'appels imbriqués, on peut épuiser l'espace disponible pour la pile !

Qu'est ce qu'il s'est passé ?

```
# let rec mklist n = match n with
| 0 -> []
| n -> n::(mklist (n-1));;
val mklist : int -> int list = <fun>
```

$$\begin{aligned} \text{mklist } 3 &\Rightarrow \underline{3::}(\text{mklist } 2) \Rightarrow \underline{3::2::}(\text{mklist } 1) \Rightarrow \underline{3::2::1}(\text{mklist } 0) \\ &\Rightarrow \underline{3::2::} \underline{1::[]} \Rightarrow \underline{3::} \underline{2::[1]} \Rightarrow \underline{3::} [2;1] \Rightarrow [3;2;1] \end{aligned}$$

- à chaque appel de fonction, il faut stocker plusieurs informations d'environnement
- ces informations sont sauvegardées dans la **pile d'exécution** (angl. **call stack**)
- quand l'appel est terminé, ces informations sont balayées du sommet de la pile

Consequence: si trop d'appels imbriqués, on peut épuiser l'espace disponible pour la pile !

Récurrance terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
 - **appel terminal**: une fonction f appelle une fonction g , mais le résultat envoyé par g est envoyé tout de suite par f ,
 - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
 - l'espace sur la pile d'exécution peut être libérée avant de lancer g !
- **Récurrance terminale** (angl.: **tail recursion**): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrance terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
 - **appel terminal**: une fonction f appelle une fonction g , mais le résultat envoyé par g est envoyé tout de suite par f ,
 - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
 - l'espace sur la pile d'exécution peut être libérée avant de lancer g !
- **Récurrance terminale** (angl.: **tail recursion**): fonction recursive, avec tous les appels rékursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrance terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
 - **appel terminal**: une fonction f appelle une fonction g , mais le résultat envoyé par g est envoyé tout de suite par f ,
 - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
 - l'espace sur la pile d'exécution peut être libérée avant de lancer g !
- **Récurrance terminale** (angl.: **tail recursion**): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrance terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
 - **appel terminal**: une fonction f appelle une fonction g , mais le résultat envoyé par g est envoyé tout de suite par f ,
 - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
 - l'espace sur la pile d'exécution peut être libérée avant de lancer g !
- **Récurrance terminale** (angl.: **tail recursion**): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrance terminale

- Il est parfois possible de réécrire une fonction en une fonction qui utilise des appels terminaux:
 - **appel terminal**: une fonction f appelle une fonction g , mais le résultat envoyé par g est envoyé tout de suite par f ,
 - dans ce cas, on n'a plus besoin des valeurs d'environnement de la fonction f quand on lance g
 - l'espace sur la pile d'exécution peut être libérée avant de lancer g !
- **Récurrance terminale** (angl.: **tail recursion**): fonction recursive, avec tous les appels récursifs à des positions terminales.
- Avantage: peu importe la profondeur de la récurrence, l'utilisation d'espace reste constante !
- En plus, le compilateur souvent optimise le temps d'exécution sur les appels terminaux !

Récurrance terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with  
| 0 -> []  
| n -> n :: (mklist (n-1));;
```

- on exécute un calcul ($n :: \dots$) avant de renvoyer le résultat de l'appel récursif ($\text{mklist } (n-1)$)
- Voici une variante terminale:

```
# let mklist n =  
  let rec mkaux n l = match n with  
  | 0 -> l  
  | n -> mkaux (n-1) (n::l)  
  in List.rev (mkaux n []);;  
val mklist : int -> int list = <fun>
```

- aucun calcul après l'appel récursif ($\text{mkaux } (n-1) (n::l)$)
- on utilise une **fonction auxiliaire** (mkaux) avec un **argument accumulateur** (l)
- attention à renverser l'ordre des elements de la liste

Récurrance terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with  
| 0 -> []  
| n -> n :: (mklist (n-1));;
```

- on exécute un calcul ($n :: \dots$) avant de renvoyer le résultat de l'appel récursif ($\text{mklist } (n-1)$)
- Voici une variante terminale:

```
# let mklist n =  
  let rec mkaux n l = match n with  
  | 0 -> l  
  | n -> mkaux (n-1) (n::l)  
  in List.rev (mkaux n []);;  
val mklist : int -> int list = <fun>
```

- aucun calcul après l'appel récursif ($\text{mkaux } (n-1) (n::l)$)
- on utilise une **fonction auxiliaire** (mkaux) avec un **argument accumulateur** (l)
- attention à renverser l'ordre des éléments de la liste

Récurrance terminale (exemples)

- Ceci n'est pas une récurrence terminale:

```
# let rec mklist n = match n with  
| 0 -> []  
| n -> n :: (mklist (n-1));;
```

- on exécute un calcul ($n :: \dots$) avant de renvoyer le résultat de l'appel récursif ($\text{mklist } (n-1)$)
- Voici une variante terminale:

```
# let mklist n =  
  let rec mkaux n l = match n with  
  | 0 -> l  
  | n -> mkaux (n-1) (n::l)  
  in List.rev (mkaux n []);;  
val mklist : int -> int list = <fun>
```

- aucun calcul après l'appel récursif ($\text{mkaux } (n-1) (n::l)$)
- on utilise une **fonction auxiliaire** (mkaux) avec un **argument accumulateur** (l)
- attention à renverser l'ordre des elements de la liste

Récurrance terminale (exemples)

```
# let mklist n =  
  let rec mkaux n l = match n with  
    | 0 -> l  
    | n -> mkaux (n-1) (n::l)  
  in List.rev (mkaux n []);;  
val mklist : int -> int list = <fun>
```

$$\begin{aligned} \text{mklist } 3 &\Rightarrow \text{List.rev(mkaux 3 [])} \Rightarrow \text{List.rev(mkaux 2 (3::[]))} \\ &\Rightarrow \text{List.rev(mkaux 2 (3::[]))} \Rightarrow \text{List.rev(mkaux 2 [3])} \\ &\Rightarrow \text{List.rev(mkaux 1 (2::[3]))} \Rightarrow \text{List.rev(mkaux 1 (2::[3]))} \\ &\Rightarrow \text{List.rev(mkaux 1 [2;3])} \text{List.rev(mkaux 0 (1::[2;3]))} \\ &\Rightarrow \text{List.rev(mkaux 0 (1::[2;3]))} \Rightarrow \text{List.rev(mkaux 0 [1;2;3])} \\ &\Rightarrow \text{List.rev [1;2;3]} \Rightarrow \dots \Rightarrow [3; 2; 1] \end{aligned}$$

- à chaque appel récursif, la quantité d'informations d'environnement à stocker est constante
- conséquence...

Récurrance terminale (exemples)

```
# let mklist n =  
  let rec mkaux n l = match n with  
    | 0 -> l  
    | n -> mkaux (n-1) (n::l)  
  in List.rev (mkaux n []);;  
val mklist : int -> int list = <fun>
```

$\text{mklist } 3 \Rightarrow \text{List.rev}(\text{mkaux } 3 []) \Rightarrow \text{List.rev}(\text{mkaux } 2 (3::[]))$
 $\Rightarrow \text{List.rev}(\text{mkaux } 2 (3::[])) \Rightarrow \text{List.rev}(\text{mkaux } 2 [3])$
 $\Rightarrow \text{List.rev}(\text{mkaux } 1 (2::[3])) \Rightarrow \text{List.rev}(\text{mkaux } 1 (2::[3]))$
 $\Rightarrow \text{List.rev}(\text{mkaux } 1 [2;3]) \text{List.rev}(\text{mkaux } 0 (1::[2;3]))$
 $\Rightarrow \text{List.rev}(\text{mkaux } 0 (1::[2;3])) \Rightarrow \text{List.rev}(\text{mkaux } 0 [1;2;3])$
 $\Rightarrow \text{List.rev } [1;2;3] \Rightarrow \dots \Rightarrow [3; 2; 1]$

- à chaque appel récursif, la quantité d'informations d'environnement à stocker est constante
- consequence. . .

Récurrance terminale (exemples)

```
# mklist 100000;;
- : int list =
[100000; 99999; 99998; 99997; 99996; 99995; 99994; 99993; 99992; 99991;
 99990; 99989; 99988; 99987; 99986; 99985; 99984; 99983; 99982; 99981; 99980;
 99979; 99978; 99977; 99976; 99975; 99974; 99973; 99972; 99971; 99970; 99969;
 99968; 99967; 99966; 99965; 99964; 99963; 99962; 99961; 99960; 99959; 99958;
 99957; 99956; 99955; 99954; 99953; 99952; 99951; 99950; 99949; 99948; 99947;
 99946; 99945; 99944; 99943; 99942; 99941; 99940; 99939; 99938; 99937; 99936;
 99935; 99934; 99933; 99932; 99931; 99930; 99929; 99928; 99927; 99926; 99925;
 99924; 99923; 99922; 99921; 99920; 99919; 99918; 99917; 99916; 99915; 99914;
 99913; 99912; 99911; 99910; 99909; 99908; 99907; 99906; 99905; 99904; 99903;
 99902; 99901; 99900; 99899; 99898; 99897; 99896; 99895; 99894; 99893; 99892;
 99891; 99890; 99889; 99888; 99887; 99886; 99885; 99884; 99883; 99882; 99881;
 99880; 99879; 99878; 99877; 99876; 99875; 99874; 99873; 99872; 99871; 99870;
 99869; 99868; 99867; 99866; 99865; 99864; 99863; 99862; 99861; 99860; 99859;
 99858; 99857; 99856; 99855; 99854; 99853; 99852; 99851; 99850; 99849; 99848;
 99847; 99846; 99845; 99844; 99843; 99842; 99841; 99840; 99839; 99838; 99837;
 99836; 99835; 99834; 99833; 99832; 99831; 99830; 99829; 99828; 99827; 99826;
 99825; 99824; 99823; 99822; 99821; 99820; 99819; 99818; 99817; 99816; 99815;
 99814; 99813; 99812; 99811; 99810; 99809; 99808; 99807; 99806; 99805; 99804;
 99803; 99802; 99801; 99800; 99799; 99798; 99797; 99796; 99795; 99794; 99793;
 99792; 99791; 99790; 99789; 99788; 99787; 99786; 99785; 99784; 99783; 99782;
 99781; 99780; 99779; 99778; 99777; 99776; 99775; 99774; 99773; 99772; 99771;
 99770; 99769; 99768; 99767; 99766; 99765; 99764; 99763; 99762; 99761; 99760;
 99759; 99758; 99757; 99756; 99755; 99754; 99753; 99752; 99751; 99750; 99749;
 99748; 99747; 99746; 99745; 99744; 99743; 99742; 99741; 99740; 99739; 99738;
 99737; 99736; 99735; 99734; 99733; 99732; 99731; 99730; 99729; 99728; 99727;
 99726; 99725; 99724; 99723; 99722; 99721; 99720; 99719; 99718; 99717; 99716;
 99715; 99714; 99713; 99712; 99711; 99710; 99709; 99708; 99707; 99706; 99705;
 99704; 99703; 99702; ...]
```

Exercices

Transformez les récurrences suivantes en récurrences terminales:

```
let rec fact n = match n with  
  | 0 -> 1  
  | n -> n*(fact (n-1));;
```

```
val fact : int -> int = <fun>
```

```
let divisors x =  
  let rec aux x c =  
    if c = x then [ x ]  
    else if (x mod c = 0) then (c::(aux x (c+1)))  
    else aux x (c+1)  
  in aux x 1;;
```

```
val divisors : int -> int list = <fun>
```

```
let rec fib n = match n with  
  | 0 | 1 -> n  
  | n -> fib(n-1)+fib(n-2);;
```

```
val fib : int -> int = <fun>
```

Recursion terminale et **try ... with**

```
# let rec copy_lines ci co =  
  try  
    let x = input_line ci  
    in  
    output_string co x;  
    output_string co "\n";  
    copy_lines ci co  
  with  
    End_of_file -> ();;  
val copy_lines : in_channel -> out_channel -> unit = <fun>  
  
# let copy infile outfile =  
  let ci = open_in infile  
  and co = open_out outfile  
  in  
    copy_lines ci co;  
    close_in ci;  
    close_out co;;  
val copy : string -> string -> unit = <fun>
```

Recursion terminale et **try ... with**

- sur des petits fichiers copy marche bien, mais en general...

```
# copy "words" "toto";;
```

```
Stack overflow during evaluation (looping recursion?).
```

- un appel récursif cesse d'être terminal si à l'intérieur d'un try:

```
try
```

```
....
```

```
copy_lines ci co
```

```
with
```

```
.....
```

copy_lines ci co n'est pas terminal car il faut mémoriser la position de son appel pour pouvoir traiter une exception éventuelle.

Recursion terminale et **try ... with**

- sur des petits fichiers copy marche bien, mais en general...

```
# copy "words" "toto";;  
Stack overflow during evaluation (looping recursion?).
```

- un appel récursif cesse d'être terminal si à l'intérieur d'un **try**:

```
try  
....  
copy_lines ci co  
with  
.....
```

copy_lines ci co n'est pas terminal car il faut mémoriser la position de son appel pour pouvoir traiter une exception éventuelle.

copy_lines en recursion terminale

```
(*copy file: tail recursive*)
type 'a option = None | Some of 'a

let rec copy_lines ci co =
  let x =
    try Some (input_line ci) with
    | End_of_file -> None
  in
  match x with
  | Some l -> output_string co l;
              output_string co "\n";
              copy_lines ci co
  | None -> ()

let copy infile outfile =
  let ci = open_in infile and co = open_out outfile in
  copy_lines ci co;
  close_in ci ;
  close_out co
```

- Voir module [Option](#) sur le Manuel.

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 éléments d'un quel que 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 éléments 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 éléments d'un quel que type `'a` et rendant en sortie un élément de ce type `'a`

Synthèse de type d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# 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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# 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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# 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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# 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 entière (`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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# 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 entière ($x+1$), 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 entière (`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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# 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 entière ($x+1$), 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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

```
# fun x -> (x+1, x+.1.0);;
```

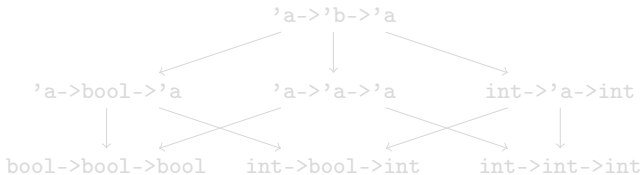
Error: This expression has **type** `int` but an expression was expected **of type** `float`

- dans la première composant de la paire, la variable `x` est utilisée dans une expression entière (`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 $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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, ...

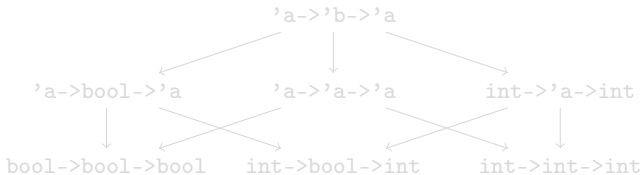


Par exemple:

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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, ...

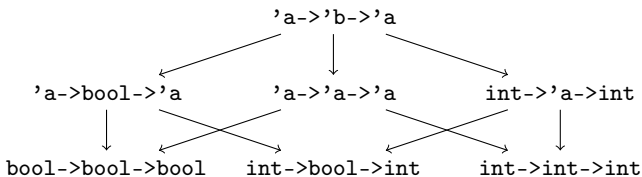


Par exemple:

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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 $'a, 'b, \dots$

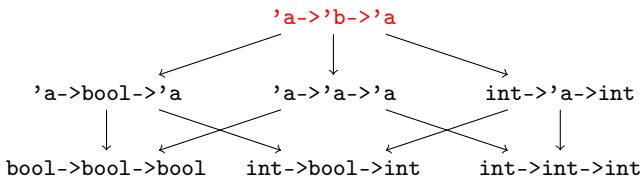


Par exemple:

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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 $'a, 'b, \dots$



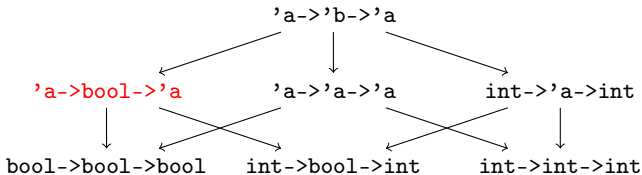
Par exemple:

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

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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 $'a, 'b, \dots$



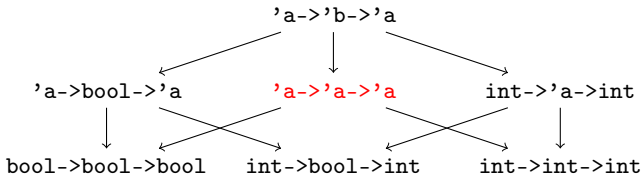
Par exemple:

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

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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, ...



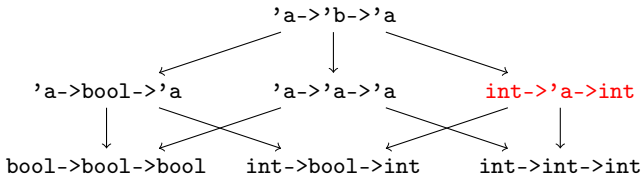
Par exemple:

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

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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, ...



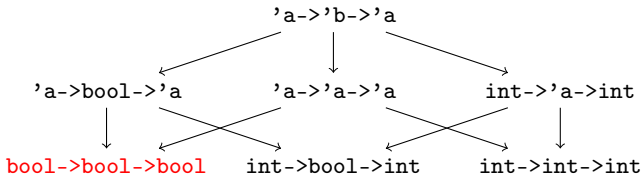
Par exemple:

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


Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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 $'a, 'b, \dots$



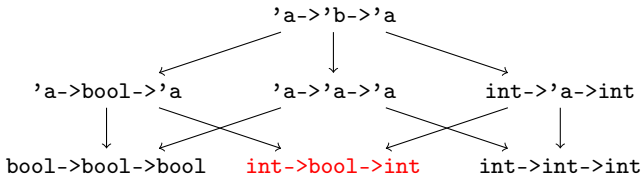
Par exemple:

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

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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, ...



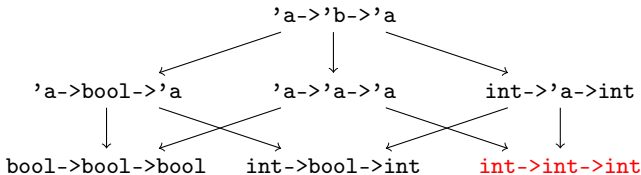
Par exemple:

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

Type principal d'une fonction

Expression $\text{expr} \longrightarrow \boxed{\text{Typeur}} \longrightarrow \text{un type } \tau \text{ pour expr}$

- 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 (permet 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 (permet 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 (permet 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 (permet 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>
```


Inférence de Type (Exercise)

Qu'affiche le toplevel Caml quand on entre les expressions suivantes ?

- ❶ `let f x g = if x then g else (fun x -> x);;`
- ❷ `let f x = if x then (fun x->x) else (fun y-> x);;`
- ❸ `let f x = if x then true else (print_string "false");;`
- ❹ `(1,"two",print_string "three");;`
- ❺ `let rec mi g l = match l with
 [] -> []
 | h::t -> (g 0 h):: (mi (fun x -> g (x+1)) t);;`
- ❻ `mi (+) [2;3;1];;`
- ❼ `let rec a = ref 0
and f () =
 let b = ref 0 in
 a := !a+1; b := !b+1; !a * !b ;;`
- ❽ `let x1 = f() in let x2 = f() in let x3 = f() in [x1;x2;x3];;`

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 contraintes que 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 types particuliers et de trouver un type plus générale:
    - ① `(f true)` force que `f` 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 contraintes que 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 types particuliers et de trouver un type plus générale:
 - ① `(f true)` force que `f` 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 contraintes que 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 types particuliers et de trouver un type plus générale:
    - ① `(f true)` force que `f` soit de type `bool->'a`
    - ② `2` force que `f` soit de type `int ->'a`

## Les limites du polymorphisme en OCaml

- Une application partielle sur un type polymorphe peut générer un **polymorphisme faible**:

```
let map_id = List.map (fun x -> x);;
val map_id : 'a list -> 'a list = <fun>
map_id [1];;
- : int list = [1]
map_id;;
- : int list -> int list = <fun>
```

- ① 'a peut être lu “pour un certain type alpha...”
- ② 'a représente un type pas encore connu, mais qu’une fois instancié, il ne sera plus générique

Ce cas se résout en explicitant le paramètre de la fonction:

```
let map_id l = List.map (fun x -> x) l;;
val map_id : 'a list -> 'a list = <fun>
```

- On rencontre le polymorphisme faible lorsque on traite aussi les structures mutables (références, tableaux, etc).

## Annotation de type

- il est quand même permis 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 permis 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 permis 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