Les types Caml
Cas particulier : les listes

Exemples

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [1];;
- : int list = [1]
# ['a'; 'b'; 'c'];;
- : char list = ['a'; 'b'; 'c']
# [ ];;
- : 'a list = [ ]
# [1; 'a'];;
This expression has type char but is here used with type
    int
```

Les types Caml
Cas particulier : les listes

Constructeurs:

[ ] et :: *(infixe)*

```
#6 :: [3; 1];;
- : int list = [6; 3; 1]

# [ ];;
- : 'a list = [ ]
```

Filtrage

```
match l with
                [ ]      -> e1
              | a : : l'  -> e2  ;;
```

Les types Caml
Cas particulier : les listes

Concaténation

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

Inverse *(version naïve)*

```
# let rec naive_rev  l = match     l      with
                                  [ ]    ->    [ ]
                              | a :: tl    -> naive_rev tl % [a];;
val naive_rev : 'a list -> 'a list = <fun>
```

Quelles sont les complexités?

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5]
naive_rev[5]
naive_rev [ ]

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

```
naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5]
naive_rev[5]
naive_rev [ ]  = [ ]
```

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5]
naive_rev[5]
naive_rev [ ] = [ ]
[ ] % [5] = [5]

0

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5]
naive_rev[5] = [5] 0

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5]
naive_rev[5] = [5] 0
[5] % [4] =[5; 4] 1

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5] = [5; 4]

0+1

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5]
naive_rev[4; 5] = [5; 4] 0+1
[5; 4] % [3] =[5; 4; 3] 2

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5] = [5; 4; 3] 0+1+2

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5]
naive_rev[3; 4; 5] = [5; 4; 3] 0+1+2
[5; 4; 3] % [2] =[5; 4; 3; 2] 3

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5] = [5; 4; 3; 2] 0+1+2+3

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5]
naive_rev [2; 3; 4; 5] = [5; 4; 3; 2] 0+1+2+3
[5; 4; 3; 2] % [1] =[5; 4; 3; 2; 1] 4

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5] = [5; 4; 3; 2; 1] 0+1+2+3+4

Les types Caml
Cas particulier : les listes

Inverse *(version naïve)*

naive_rev [1; 2; 3; 4; 5] = [5; 4; 3; 2; 1] 0+1+2+3+4

Coût dans le cas d'une liste de longueur *n* :

$1 + 2 \ldots + (n-1) = n(n-1)/2 \in \Theta(n^2)$

Algorithme quadratique

Les types Caml
Cas particulier : les listes

Inverse *(version correcte)*

[1]
[2; 1]
[3; 2; 1]
[4; 3; 2; 1]
[5; 4; 3; 2; 1]

[1; 2; 3; 4; 5]
[2; 3; 4; 5]
[3; 4; 5]
[4; 5]
[5]
[ ]

Les types Caml
Cas particulier : les listes

Inverse *(version correcte)*

[ ]
[1]
[2; 1]
[3; 2; 1]
[4; 3; 2; 1]
[5; 4; 3; 2; 1]

[1; 2; 3; 4; 5]
[2; 3; 4; 5]
[3; 4; 5]
[4; 5]
[5]
[ ]

Les types Caml
Cas particulier : les listes

Inverse *(version correcte)*

[ ]	[1; 2; 3; 4; 5]
[1]	[2; 3; 4; 5]
[2; 1]	[3; 4; 5]
[3; 2; 1]	[4; 5]
[4; 3; 2; 1]	[5]
[5; 4; 3; 2; 1]	[ ]

D'où l'idée d'une fonction *rev'* opérant sur 2 listes
Ôter la tête de la 2ème et la rajouter en tête de la 1ère

Complexité?

Les types Caml
Cas particulier : les listes

Inverse *(version correcte)*

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

# rev' [1; 2; 3; 4; 5] [6; 7; 8; 9];;
- : int list = [9; 8; 7; 6; 1; 2; 3; 4; 5]

# let rev = rev' [ ];;
val rev : '_a list -> '_a list = <fun>

# rev  [9; 8; 7; 6; 1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1; 6; 7; 8; 9]
```

Les types Caml
Cas particulier : les listes

Inverse *(version correcte)*

```
# let rev =
            let rec rev' l1 l2 = match   l2   with
                                          [ ]    -> l1
                                        |a :: tl  -> rev' (a :: l1) tl
             in       rev' [ ];;
val rev : '_a list -> '_a list = <fun>

# rev  [9; 8; 7; 6; 1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1; 6; 7; 8; 9]
```

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

```
List.fold_right  : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.fold_left   : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

récursion terminale: les calculs se font au fur et à mesure sans rien empiler

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

récursion non terminale: on empile a_1, ..., a_n avant de pouvoir commencer à calculer la première valeur de f.

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

(%) [a1; a2; ...; an] l2 = a1 :: (a2 :: (... (an :: l2)))

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

(%) [a1; a2; ...; an] l2 = a1 :: (a2 :: (... (an :: l2)))

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

(%) [a1; a2; ...; an] l2 = a1 :: (a2 :: (... (an :: l2)))

e = l2 et f a l = a :: l

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

(%) [a1; a2; ...; an] l2 = a1 :: (a2 :: (... (an :: l2)))

e = l2 et f a l = a :: l

(%) l1 l2 = List.fold_right (fun a l -> a::l) l1 l2;;

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

(%) [a1; a2; ...; an] l2 = a1 :: (a2 :: (... (an :: l2)))

e = l2 et f a l = a :: l

(%) l1 l2 = List.fold_right (fun a l -> a::l) l1 l2;;

(%) = List.fold_right (fun a l -> a::l) ;;

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

rev [a1; a2; ...; an] = an :: (... (a2 :: (a1 :: [ ])))

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

rev [a1; a2; ...; an] = an :: (... (a2 :: (a1 :: [ ])))

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

rev [a1; a2; ...; an] = an :: (... (a2 :: (a1 :: [ ])))

e = [ ] et (f l a) = a :: l

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

rev [a1; a2; ...; an] = an :: (... (a2 :: (a1 :: [ ])))

e = [ ] et (f l a) = a :: l

```
rev l = List.fold_left (fun l a -> a :: l) [ ] l ;;
```

Les types Caml
Cas particulier : les listes

Itérateurs sur les listes

fold_left f e [a1; a2; ...; an] = (f ... (f (f e a1) a2) ... an)

fold_right f [a1; a2; ...; an] e = (f a1 (f a2 ... (f an e)))

rev [a1; a2; ...; an] = an :: (... (a2 :: (a1 :: [ ])))

e = [ ] et (f l a) = a :: l

rev l = List.fold_left (fun l a -> a :: l) [ ] l ;;

rev = List.fold_left (fun l a -> a :: l) [ ];;

Les types Caml
Les types sommes

Constructeurs constants: *Types énumérés*

type *nom* = Nom_1 | ... | Nom_n

```
# type traffic_light = Red | Green | Orange;;
type traffic_light = Red | Green | Orange

# Red;;
- : traffic_light = Red
```

Constructeurs: Red, Green, Orange

Leur nom débute obligatoirement par une majuscule

Les types Caml
Les types sommes

Constructeurs constants: *Types énumérés*

type *nom* = Nom_1 | ... | Nom_n

```
# type traffic_light = Red | Green | Orange;;
type traffic_light = Red | Green | Orange
```

Constructeurs: Red, Green, Orange

Le type traffic_light est une l'union de ses constructeurs

traffic_light = {Red} U {Green} U {Orange} = {Red, Green, Orange}

Les types Caml
Les types sommes

Constructeurs constants

Types énumérés: le type unit

Le type prédéfini *unit* a un seul élément : ( )

```
# ( );;
- : unit = ( )
```

Les types Caml
Les types sommes

Constructeurs constants

Types énumérés: autre exemple

```
# type boolean = T | F;;
type boolean = T | F
```

Filtrage sur constructeurs

```
# let ou b = match b  with
              T  ->  (fun x -> T)
            | F  ->  (fun x -> x);;
val ou : boolean -> boolean -> boolean = <fun>
```

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
```

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
```

Cela permet de représenter l'union des entiers et des flottants:

int U float

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
```

Cela permet de représenter l'union des entiers et des flottants:

int ∪ *float*

num = {Int x / x:int } ∪ {Fl x / x: float}

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
```

num = {Int x / x:int } ∪ {Fl x / x: float}

```
# Int (10/3);;
- : num = Int 3
```

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
```

num = { Int x / x:int } U { Fl x / x: float}

```
# Int (10/3);;
- : num = Int 3
# Fl (10./.3.);;
- : num = Fl 3.3333333333333348
```

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
```

Exemple de filtrage

```
# let add_num x y = match (x,y) with
  (Int x1, Int x2)   ->    Int (x1 + x2)
  | (Fl x1,  Fl  x2)    ->   Fl  (x1 +. x2)
  | (Int x1, Fl  x2)   ->    Fl (float_of_int x1 +. x2)
  | (Fl x1,  Int x2)    ->   Fl (x1 +. float_of_int x2);;
val add_num : num -> num -> num = <fun>
```

Les types Caml
Les types sommes

Constructeurs avec arguments

type *nom* = Nom_1 of $type_1$ | ... | Nom_n of $type_n$

Si $type_i$ est omis, Nom_i est une constante du type défini

Exemple

Extension d'un type à une valeur exceptionnelle

```
# type extend_float = None | Some of float;;
type extend_float = None | Some of float
```

extend_float = {None} ∪ {Some x / x: float}

Utilisé pour traiter un cas exceptionnel en fonctionnel pur

Les types Caml
Les types sommes

Constructeurs avec arguments

```
# type extend_float = None | Some of float;;
type extend_float = None | Some of float
```

Exemple: *plus grande des 2 racines de* $ax^2 + bx + c$

```
# let max_root a b c =
if a<>0. then
   let delta = b*.b -. 4.*.a *.c in
       if delta <= 0.      then  None
       else          if a>0. then  Some ((-.b +. sqrt delta)/.2.*.a)
                     else          Some ((-.b -. sqrt delta)/.2.*.a)
else  None ;;
val max_root: float->float->float->extend_float = <fun>
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Type à un seul constructeur

Le nouveau type est isomorphe au type de base

```
# type age = Old of int;;
  type age = Old of int
```

```
# type date = Year of int;;
  type date = Year of int
```

Les deux types sont isomorphes à *int* mais permettent de distinguer un *âge* d'une *date*

```
# Old 40;;
- : age = Old 40
```

```
# Year 1996;;
- : date = Year 1996
```

Les types Caml
Les types sommes

Constructeurs avec arguments
Types récursifs

Le nom du type apparaît dans un au moins des $type_i$

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

# let rec length l = match l with
                       Nil -> 0
                   | Cons(_, tl) -> length tl+1;;
val length : list -> int = <fun>
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Types récursifs: exemple des arbres binaires

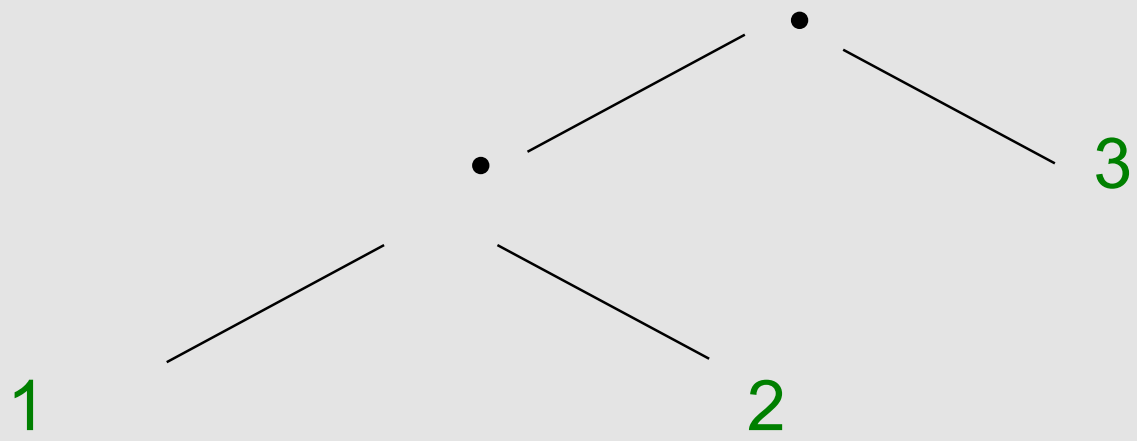
```
# type int_tree = Leaf of int | Node of int_tree * int_tree;;
type int_tree = Leaf of int | Node of int_tree * int_tree
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Types récursifs: exemple des arbres binaires

```
# type int_tree = Leaf of int | Node of int_tree * int_tree;;
type int_tree = Leaf of int | Node of int_tree * int_tree
```

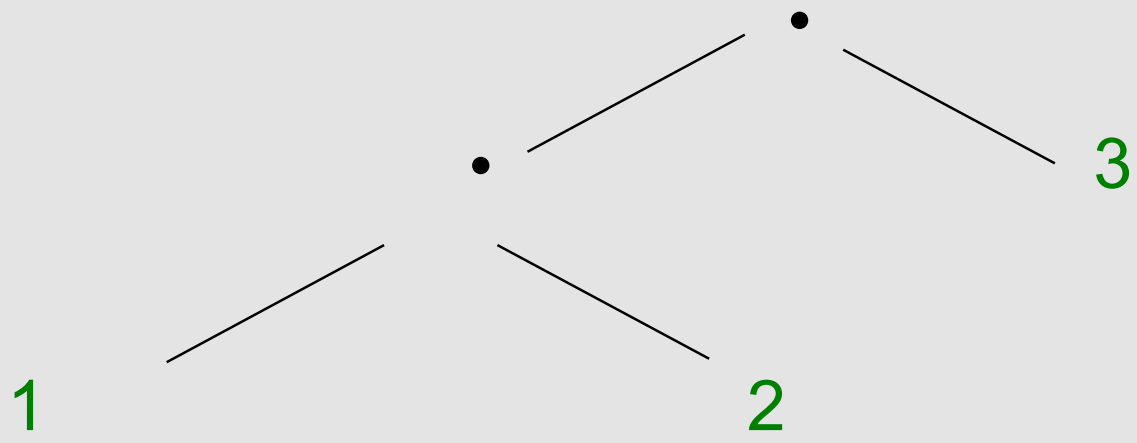


Les types Caml
Les types sommes

Constructeurs avec arguments

Types récursifs: exemple des arbres binaires

```
# type int_tree = Leaf of int | Node of int_tree * int_tree;;
type int_tree = Leaf of int | Node of int_tree * int_tree
```




```
# Node (Node (Leaf 1, Leaf 2), (Leaf 3));;
- : int_tree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Types récursifs: exemple des arbres binaires

```
# let rec size t = match t with
      Leaf _       ->  1
   | Node (sag, sad)  ->  size sag + size sad + 1;;
val size : int_tree -> int = <fun>

# size (Node (Node (Leaf 1, Leaf 2), (Leaf 3)));;
- : int = 5
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Types polymorphes

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree

# size (Node (Node (Leaf 1, Leaf 2), (Leaf 3)));;
- : int = 5

# size (Node (Node (Leaf “chou”, Leaf “pou”),  (Leaf “hibou”)));;
- : int = 5
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Types polymorphes

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

```
# type intTree = int tree;;
type intTree = int tree
```

Définition d'un synomyme

Les types Caml
Les types sommes

Constructeurs avec arguments

Types polymorphes

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

```
# type intTree = int tree;;
type intTree = int tree
```

Définition d'un synomyme

```
# Node (Node (Leaf 1, Leaf 2), (Leaf 3));;
- : int tree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
```

Les types Caml
Les types sommes

Constructeurs avec arguments

Types polymorphes

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
```

```
# type intTree = int tree;;
type intTree = int tree
```

Définition d'un synomyme

```
# ( Node (Node (Leaf 1, Leaf 2), (Leaf 3)) : intTree );;
- : intTree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
```

Les types Caml
Les types sommes

Contraintes de types: *Autre exemple*

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

Les types Caml
Les types sommes

Contraintes de types: *Autre exemple*

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

Restriction de *comp* à des fonctions
g : int ->bool et *f: bool -> bool*

Les types Caml
Les types sommes

Contraintes de types: *Autre exemple*

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

Restriction de *comp* à des fonctions
g : int -> bool et *f: bool -> bool*

```
# let comp (f: bool->bool) (g: int->bool) x =  f (g x);;
val comp:(bool -> bool)->(int -> bool)-> int ->bool = <fun>
```

Les types Caml
Les types sommes

Contraintes de types: *Autre exemple*

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

Restriction de *comp* à des fonctions
g : int -> bool et *f: bool -> bool*

```
# let comp:(bool -> bool) -> (int -> bool) -> int -> bool =
                                        fun f g x -> f (g x);;
val comp:(bool -> bool)-> (int -> bool) -> int -> bool = <fun>
```

Retour sur le filtrage

Motif: *construction syntaxique de la forme*

motif ::= variable
| _
| (motif)
| constante
| constructeur motif
| (motif, ... , motif)
| [ ]
| motif :: motif
| [motif; ... ; motif]

Motif *linéaire*: 1 seule occurrence de chaque variable

Exemple: M(x,y,a,t) = [(x, "chou" , [ ]) ; (a :: y, t, [1; 2]) ; _]

Retour sur le filtrage

Qu'est-ce que le filtrage ?

Evaluation de match v with $M(x_1, ..., x_n)$

Données

une valeur v

un motif linéaire $M(x_1, ..., x_n)$ de variables $x_1, ..., x_n$

Résultat: une substitution s de $x_1, ..., x_n$ telle que

$$v = M (s(x_1), ... , s(x_n))$$

Ou encore

trouver $a_1 , ..., a_n$ tels que si $s(x_1) = a_1$ ... $s(x_n) = a_n$ alors

$$v = M(a, ... , a_n)$$

Retour sur le filtrage

Exemple

Valeur

[([1;2],“chou”,[ ]); ([5], “bijou”, [1;2]); ([0;1],“pou”,[ ])]

Motif M(x, y, a, t)

[(x, “chou”, [ ]) ; (a :: y, t, [1; 2]) ; _]

Résultat

s(x) = [1; 2]
s(a) = 5
s(y) = [ ]
s(t) = “bijou”
s(u) = ([0;1],“pou”, [ ])

s = [(x, [1; 2]); (a, 5); (y, [ ]); (t, “bijou”); (u,([0;1],“pou”,[ ]) *]*

Retour sur le filtrage

Cas particulier

un filtrage avec un seul motif *équivaut à un* let in

match *expr* with *motif* -> *expr'* <=>
let *motif* = *expr* in *expr'*

Retour sur le filtrage

Cas particulier

un filtrage avec un seul motif *équivaut à un* let in

match *expr* with *motif* -> *expr'* <=>
let *motif* = *expr* in *expr'*

```
match  milieu p p' with  (x, y) ->  x+y
let (x, y) = milieu p p'  in x+y
```

Retour sur le filtrage

Cas particulier

un filtrage avec un seul motif *équivaut à un* let in

```
match expr with motif -> expr'   <=>
let motif = expr in expr'
```

```
match  milieu p p' with  (x, y) ->  x+y
let (x, y) = milieu p p'  in x+y
```

```
match moyenneDage l with   Old n -> n
let Old n = moyenneDage l  in n
```

Retour sur le filtrage

Cas particulier

un filtrage avec un seul motif *équivaut à un* let in

Une définition est un cas particulier de filtrage:
l'expression filtrée est une variable.

let f *motif* = *expr* let f x = match x with *motif* -> *expr*

Exemple:

```
# let fst (x, y) = x;;
# let fst c = match c with (x, y) -> x;;
```

Retour sur le filtrage

Exemple

```
# type point = float * float;;
# let (centre_de_gravite: point -> point -> point -> point) =
fun (x1, y1) (x2, y2) (x3, y3) ->
  ((x1 +. x2 +. x3) /. 3. , (y1 +. y2 +. y3) /. 3.);;
```

Retour sur le filtrage

Exemple

```
# type point = float * float;;
# let (centre_de_gravite: point -> point -> point -> point) =
fun (x1, y1) (x2, y2) (x3, y3) ->
  ((x1 +. x2 +. x3) /. 3. , (y1 +. y2 +. y3) /. 3.);;

# centre_de_gravite  (1.,1.) (2.,2.) (3.,3.);;
- : point = (2., 2.)
```

Retour sur le filtrage

Exemple

```
# type point = float * float;;
# let (centre_de_gravite: point -> point -> point -> point) =
fun (x1, y1) (x2, y2) (x3, y3) ->
  ((x1 +. x2 +. x3) /. 3. , (y1 +. y2 +. y3) /. 3.);;

# centre_de_gravite  (1.,1.) (2.,2.) (3.,3.);;
- : point = (2., 2.)

# centre_de_gravite (0.,0.) (1.,1.) (-1., -1.);;
- : point = (0., 0.)
```

Retour sur le filtrage

Exemple

```
# let (centre_de_gravite: point -> point -> point -> point) =
fun (x1, y1) (x2, y2) (x3, y3) ->
  ((x1 +. x2 +. x3) /. 3. , (y1 +. y2 +. y3) /. 3.);;
```

Isobarycentre de 6 points : *le milieu des centres de gravité pris 3 à 3*

Retour sur le filtrage

Exemple

```
# let (centre_de_gravite: point -> point -> point -> point) =
fun (x1, y1) (x2, y2) (x3, y3) ->
  ((x1 +. x2 +. x3) /. 3. , (y1 +. y2 +. y3) /. 3.);;
```

Isobarycentre de 6 points :

le milieu des centres de gravité pris 3 à 3

```
# let iso_barycentre a b c a' b' c' =
let (x, y) =  centre_de_gravite  a b c  in
    let (x', y') = centre_de_gravite a' b' c' in
                  ( ((x+.x')/.2. , (y +. y')/.2.) :point) ;;
```

Retour sur le filtrage

Exemple

```
# let iso_barycentre a b c a' b' c'=
let (x, y) =  centre_de_gravite  a b c  in
   let (x', y') = centre_de_gravite a' b' c' in
                  ( ((x+.x')/.2. , (y +. y')/.2.) :point) ;;

val iso_barycentre :
  point ->point ->point ->point ->point ->point = <fun>
```

Retour sur le filtrage

Exemple

```
# let iso_barycentre a b c a' b' c'=
let (x, y) =  centre_de_gravite  a b c  in
   let (x', y') = centre_de_gravite a' b' c' in
                ( ((x+.x')/.2. , (y +. y')/.2.) :point) ;;
```

```
# let iso_barycentre a b c a' b' c'=
match  centre_de_gravite  a b c  with
  (x, y) -> match centre_de_gravite a' b' c' with
            (x', y') -> ( ((x+.x')/.2. , (y +. y')/.2.) :point) ;;
```

Les types Caml
Les types sommes

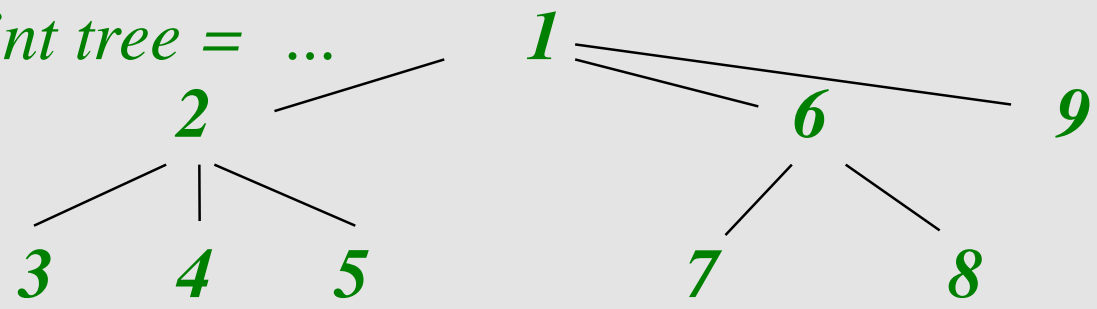
Arbres quelconques

```
#  type 'a tree = Tr of 'a*'a tree list;;
type 'a tree = Tr of 'a * 'a tree list

#let my_tree = (Tr(1,
                   [Tr (2, [Tr (3, []); Tr (4, []); Tr (5, [])]);
                    Tr (6, [Tr (7, []); Tr (8, [])]);
                    Tr (9, [ ])]));;
val my_tree : int tree =  ...
```

Les types Caml
Les types sommes

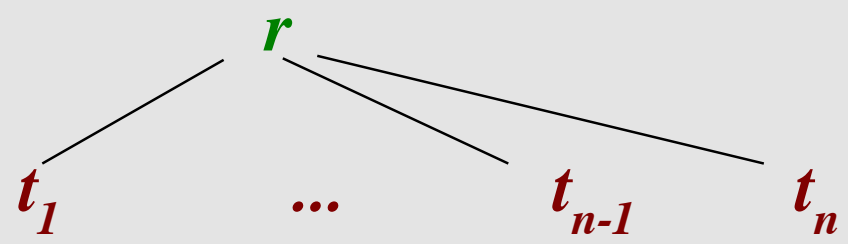
Arbres quelconques

```
#  type 'a tree = Tr of 'a*'a tree list;;
type 'a tree = Tr of 'a * 'a tree list

#let my_tree = (Tr(1,
                   [Tr (2, [Tr (3, []); Tr (4, []); Tr (5, [])]);
                    Tr (6, [Tr (7, []); Tr (8, [])]);
                    Tr (9, [ ])]));;
val my_tree : int tree =  ...
```



Les types Caml
Les types sommes

Arbres quelconques

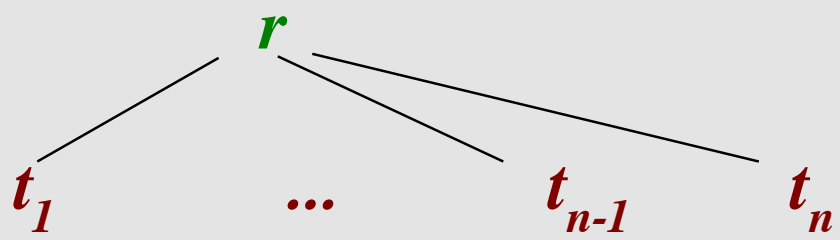


$$size\ Tr(r, [t_1; \ldots ; t_n]) = 1 + \underline{size\ t_1 + \ldots + size\ t_n}$$

racine — fils

Les types Caml
Les types sommes

Arbres quelconques

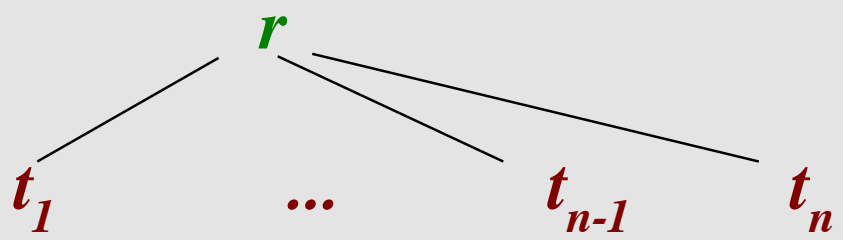


$$size\ Tr(r, [t_1; \ldots ; t_n]) = 1 + size\ t_1 + \ldots + size\ t_n$$

Parcours des fils de gauche à droite

Les types Caml
Les types sommes

Arbres quelconques



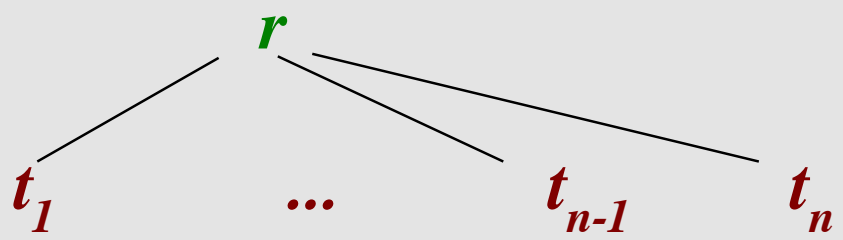
$$size\ Tr(r, [t_1; \ldots ; t_n]) = 1 + \quad size\ t_1 + \ldots + size\ t_n$$

Parcours des fils de gauche à droite

$$size\ t_1 + \ldots + size\ t_n = \text{fold\_left}\ h\ 0\ [t_1; \ \ldots; t_n]$$

Les types Caml
Les types sommes

Arbres quelconques



$$size\ Tr(r, [t_1; \ldots ; t_n]) = 1 + size\ t_1 + \ldots + size\ t_n$$

Parcours des fils de gauche à droite

size t_1 + ... + size t_n = fold_left h 0 [t_1; ...; t_n]

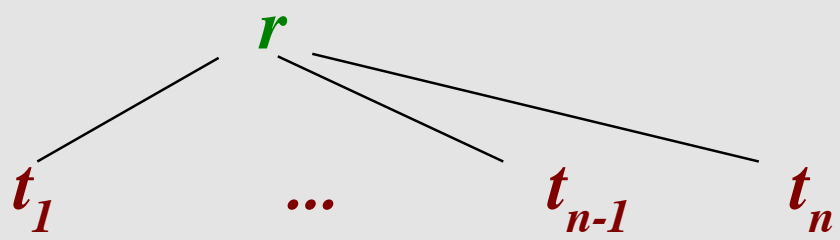
h = fun s t -> ?

s : Σ (tailles des fils déjà visités)

t : fils courant t

Les types Caml
Les types sommes

Arbres quelconques



$$size\ Tr(r, [t_1; \ldots ; t_n]) = 1 + \quad size\ t_1 + \ldots + size\ t_n$$

Parcours des fils de gauche à droite

size t_1 + ... + size t_n = fold_left h 0 [t_1; ...; t_n]

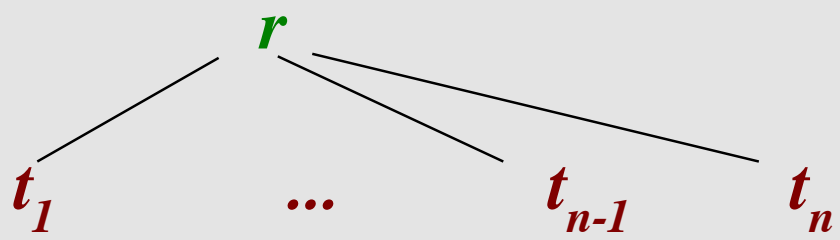
h = fun s t -> s + size t

s : Σ (tailles des fils déjà visités)

t : fils courant t

Les types Caml
Les types sommes

Arbres quelconques

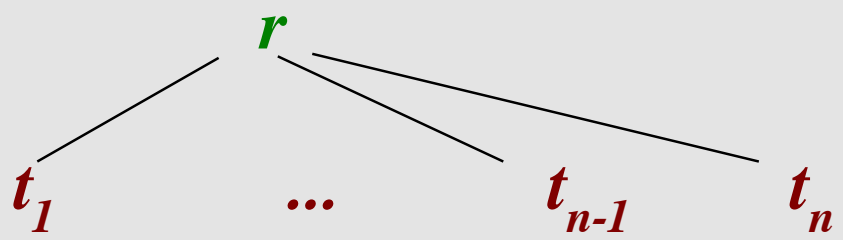


$$size\ Tr(r, [t_1; \ldots ; t_n]) = 1 + \quad size\ t_1 + \ldots + size\ t_n$$

```
#let rec size (Tr(r, l)) = 1+ List.fold_left (fun s t -> s + size t)  0 l;;
```

Les types Caml
Les types sommes

Arbres quelconques

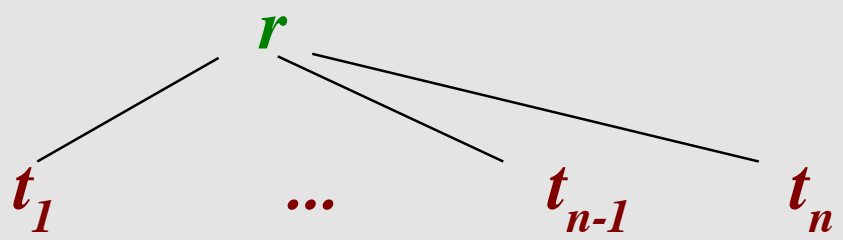


height Tr(r, [t_1; ... ; t_n])=

1 + *(max ... (max 0 (height t_1)) ...(height t_n))*

Les types Caml
Les types sommes

Arbres quelconques



height Tr(r, [t_1; ... ; t_n])=

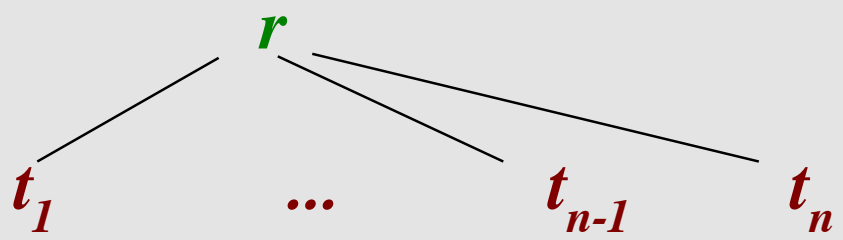
1 + *(max ... (max 0 (height t_1)) ...(height t_n))*

racine

fils

Les types Caml
Les types sommes

Arbres quelconques



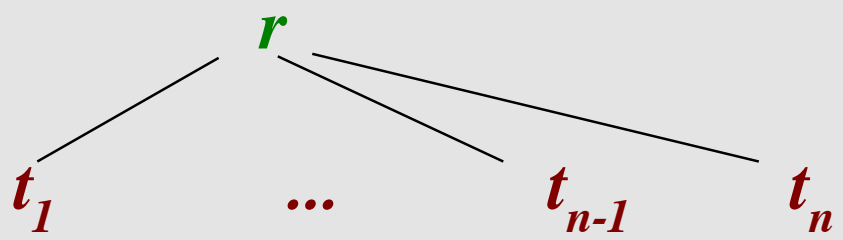
height Tr(r, [t_1; ... ; t_n])=

1 + *(max ... (max 0 (height t_1)) ...(height t_n))*

fold_left h 0 [t_1; ...; t_n]

Les types Caml
Les types sommes

Arbres quelconques



height Tr(r, [t_1; ... ; t_n])=

1 + *(max ... (max 0 (height t_1)) ...(height t_n))*

fold_left h 0 [t_1; ...; t_n]

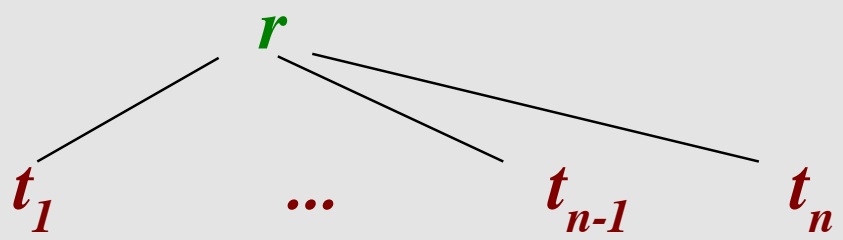
h = fun x t -> ?

↓ ↓

max (hauteurs des fils visités) fils courant

Les types Caml
Les types sommes

Arbres quelconques



height Tr(r, [t_1; ... ; t_n])=

1 + *(max ... (max 0 (height t_1)) ...(height t_n))*

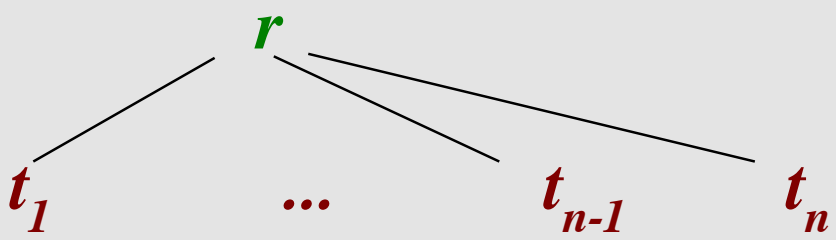
fold_left h 0 [t_1; ...; t_n]

h = fun x t -> max x (height t)

max (hauteurs des fils visités) fils courant

Les types Caml
Les types sommes

Arbres quelconques



height Tr(r, [t_1; ... ; t_n])=

1 + (max ... (max 0 (height t_1)) ...(height t_n))

```
#let rec height (Tr(r, l)) =
        1   +     List.fold_left (fun x t -> max x (height t)) 0 l ;;
```

Les types Caml
Les types sommes

Arbres quelconques *Parcours récursif gauche-droite*

$$lrt\ (Tr(r\ ,\ [t_1;\ \ldots\ ;\ t_n])) = f\ \ r\ \underbrace{(g\ \ldots\ (g\ e\ \ (lrt\ \ t_1\ ))\ \ldots(lrt\ t_n))}_{x}$$

size | height

x : contribution des fils

size	height
$x = \sum$ *tailles*	*x = max des profondeurs*
g = addition, e = 0	*g = max, e = 0*

f *: contribution de la racine*

size	height
f r x = 1 + x	f r x = 1 + x
1 noeud de plus	*1 niveau de plus*

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droite

$$lrt\ (Tr(r\ ,[t_1;\ \ldots\ ;\ t_n])) = f\ \ r\ \underbrace{(g\ \ldots\ (g\ e\ \ (lrt\ \ t_1\ ))\ \ldots(lrt\ t_n))}_{x}$$

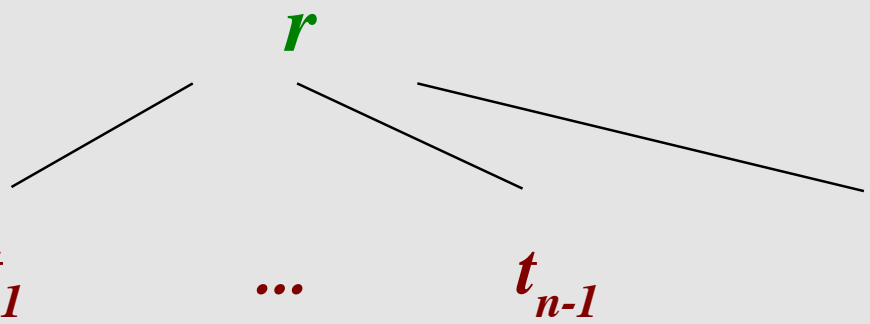
$$x = \ fold\_left\ (fun\ x\ t\ ->\ g\ x\ (lrt\ t))\quad e\quad [t_1;\ \ldots\ ;\ t_n]$$

$$lrt\ (Tr(r\ ,\ l)) = f\ r\ (fold\_left\ (fun\ x\ t\ ->\ g\ x\ (lrt\ t))\ e\ l)$$

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droit

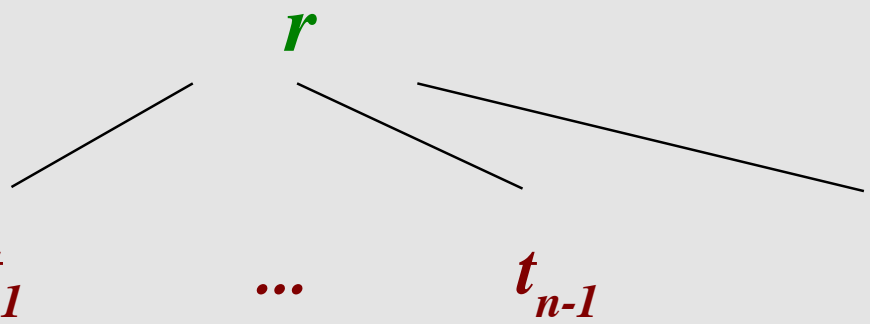


lrt (Tr(r , l)) = (f r (fold_left (fun x t -> g x (lrt t)) e l)

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droit



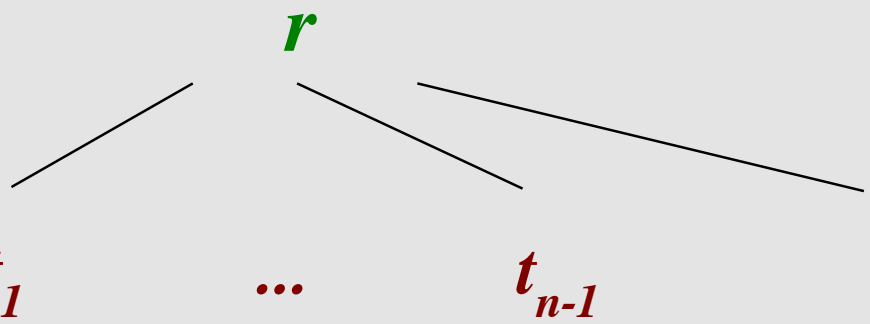
lrt (Tr(r , l)) = (f r (fold_left (fun x t -> g x (lrt t)) e l)

```
# let rec lr_traverse f g e (Tr(r, l)) =
```

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droit



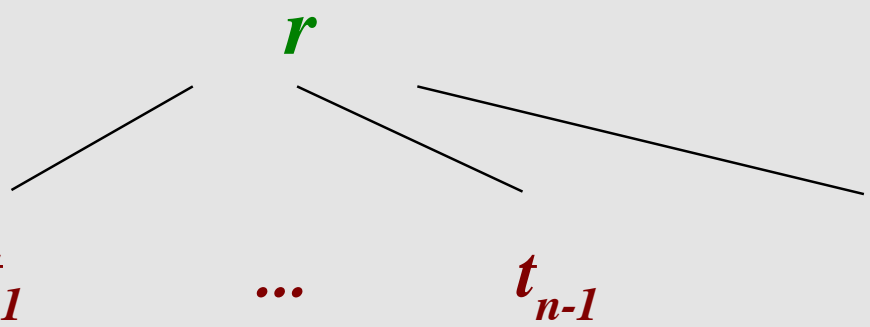
lrt (Tr(r , l)) = (f r (fold_left (fun x t -> g x (lrt t)) e l)

```
# let rec lr_traverse f g e (Tr(r, l)) =
f r (List.fold_left (fun x t->g x (lr_traverse f g e  t)) e l);;
```

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droit



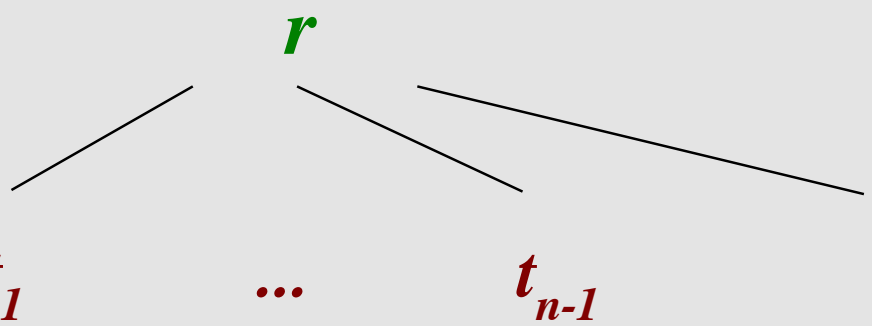
lrt (Tr(r , l)) = (f r (fold_left (fun x t -> g x (lrt t)) e l)

```
# let rec lr_traverse f g e (Tr(r, l)) =
f r (List.fold_left (fun x t->g x (lr_traverse f g e  t)) e l);;

val lr_traverse : ('a -> 'b -> 'c) -> ('b -> 'c -> 'b) -> 'b -> 'a tree
                                                    -> 'c = <fun>
```

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droit




```
lrt (Tr(r , l)) = (f r (fold_left (fun x t -> g x (lrt t)) e l)
```

```
# let rec lr_traverse f g e (Tr(r, l)) =
f r (List.fold_left (fun x t->g x (lr_traverse f g e  t)) e l);;
```

```
# let  lr_traverse f g e =
      let rec lrt (Tr(r, l)) = f r (List.fold_left (fun x t->g x (lrt  t)) e l)
           in lrt ;;
```

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif gauche-droite

```
# let  lr_traverse f g e =
      let rec lrt (Tr(r, l)) = f r (List.fold_left (fun x t->g x (lrt  t)) e l)
          in lrt ;;


#let size = lr_traverse (fun r x -> 1+ x) (+) 0;;

#let heigth = lr_traverse (fun r x -> 1+ x) max  0;;
```

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

$rlt\ \ Tr(r\ ,[t_1;\ \ldots\ ;\ t_n]) =\ f\ \ r\ \ (g\ (rlt\ t_1)\ (g\ \ldots\ (g\ (rlt\ t_n)\ e)))$

$rlt\ (Tr(r\ ,\ l)) = (f\ r\ (fold\_right\ (fun\ t\ x\ ->\ g\ (rlt\ t)\ x)\ l\ e)$

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

$rlt\ \ Tr(r\ ,[t_1;\ \ldots\ ;\ t_n]) =\ f\ \ r\ \ (g\ (rlt\ t_1)\ (g\ \ldots\ (g\ (rlt\ t_n)\ e)))$

$rlt\ (Tr(r\ ,\ l)) = (f\ r\ (fold\_right\ (fun\ t\ x\ ->\ g\ (rlt\ t)\ x)\ l\ e)$

η-réduction

$fun\ t\ x\ ->\ g\ (rlt\ t)\ x = fun\ t\ ->\ g\ (rlt\ t)$

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

rlt Tr(r ,[t_1; ... ; t_n]) = f r (g (rlt t_1) (g ... (g (rlt t_n) e)))
rlt (Tr(r , l)) = (f r (fold_right (fun t x -> g (rlt t) x) l e)

η-réduction

fun t x -> g (rlt t) x = fun t -> g (rlt t) = g % rlt

avec *# let (%) f g x = f (g x);;* la composition

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

$rlt\ \ Tr(r\ ,[t_1;\ \ldots\ ;\ t_n]) =\ f\ \ r\ \ (g\ (rlt\ t_1)\ (g\ \ldots\ (g\ (rlt\ t_n)\ e)))$

$rlt\ (Tr(r\ ,\ l)) = f\ r\ (fold\_right\ (fun\ t\ x\ ->\ g\ (rlt\ t)\ x)\ l\ e)$

η-réduction

$fun\ t\ x\ ->\ g\ (rlt\ t)\ x = fun\ t\ ->\ g\ (rlt\ t) = g\ \%\ rlt$

$rlt\ (Tr(r\ ,\ l)) = f\ r\ (fold\_right\ (g\%rlt)\ l\ e)$

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

rlt (Tr(r , l)) = f r (fold_right (g % rlt) l e)

```
#let rl_traverse f g e =
      let rec  rlt  (Tr( r, l )) = f r (List.fold_right (g % rlt) l e)
           in  rlt;;
```

```
val rl_traverse : ('a -> 'b -> 'c) -> ('c -> 'b -> 'b) -> 'b ->
                  'a tree -> 'c = <fun>
```

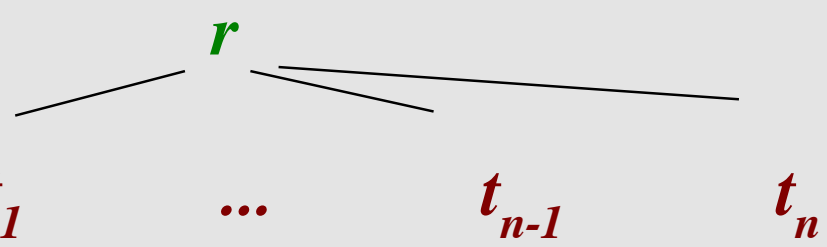
Les types Caml
Les types sommes

Arbres quelconques

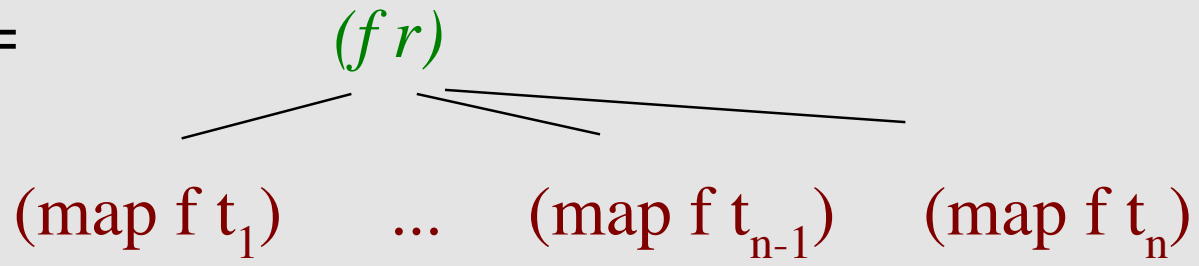
Parcours récursif droite-gauche

Application d'une fonction f à tous les noeuds d'un arbre
(map f t)

si t =



(map f t) =



Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

Application d'une fonction f à tous les noeuds d'un arbre

map f (Tr(r, [t_1 ; ...; t_n])) = Tr(f r, [map f t_1 ; .. ; map f t_n])

La liste [map f t_1 ; .. ; map f t_n]
est construite de droite à gauche (linéaire)
plutôt que de gauche à droite (quadratique)

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

Application d'une fonction f à tous les noeuds d'un arbre

map f (Tr(r, [t_1 ; ...; t_n])) = Tr(f r, [map f t_1 ; .. ; map f t_n])

```
# let map f =  rl_traverse (fun r l -> Tr (f r, l)) (fun a l -> a :: l) [  ] ;;
```

Les types Caml
Les types sommes

Arbres quelconques

Parcours récursif droite-gauche

Application d'une fonction f à tous les noeuds d'un arbre

```
# let map f  = rl_traverse (fun r l -> Tr (f r, l)) (fun a l -> a :: l)[  ]  ;;
```

```
 # map (fun x -> x*x) my_tree;;
- : int tree = Tr (1,
                   [Tr (4, [Tr (9, [ ]); Tr (16, [ ]); Tr (25,[ ])]);
                    Tr (36, [Tr (49, []); Tr (64, [])]);
                    Tr (81,[])])
```