Programmation Avancée

Types algébriques généralisés (GADTs)

Armaël Guéneau

une partie des transparents est adaptée du cours de Jeremy Yallop (Cambridge)

rappel du cours sur les modules : types fantômes

paramètres de types garantissant des invariants supplémentaires grâce à l'abstraction

```
type 'a str
(* implémentation: type 'a str = string *)
type clean
type dirty

val read : unit -> dirty str
val sanitize : dirty str -> clean str
val write : clean str -> unit
```

'a str : 'a est un **paramètre fantôme** capturant des invariants supplémentaires

aujourd'hui : GADTs

Types de données algébriques généralisés, qui associent **constructeurs de données** avec des **paramètres de types** riches

Aspect clef sous-jacent : une notion d'**égalité** entre types

ce qu'on y gagne

- on peut typer d'avantage de programmes
- on peut donner des types + riches et + précis
- dans certains cas, on gagne en performance

prix à payer

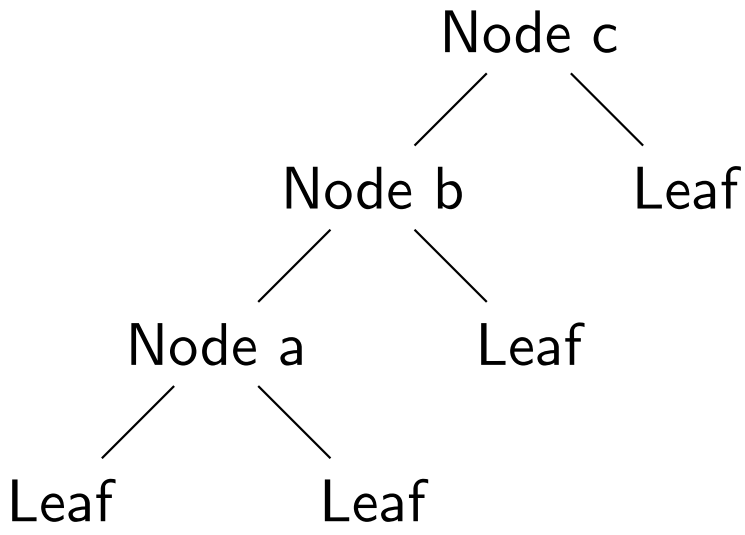
pour en profiter, on doit :

- **décrire nos types de données** plus précisement
- lier étroitement **types et données**
- voir certains programmes comme calculant des **témoins de preuve**

applications : teaser

- arbres fortement typés où le type spécifie la hauteur
- `printf` bien typé
- listes contenant des données de types différents
- représentations de types à l'exécution
- ...

exemple : arbres




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

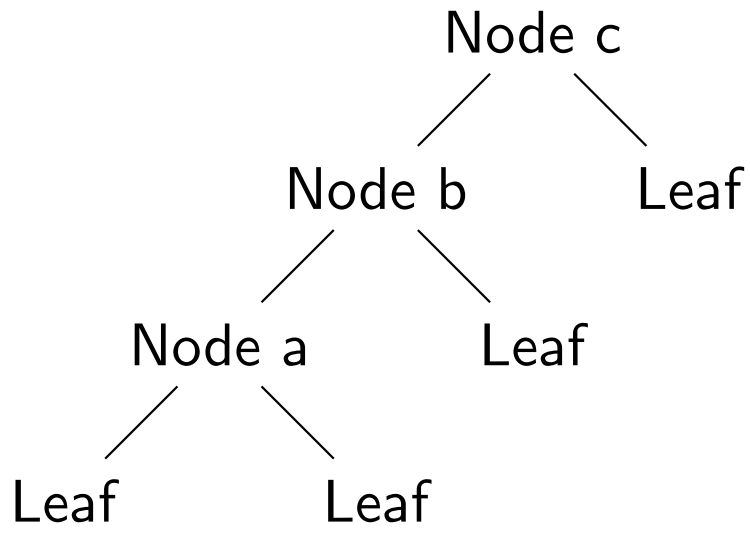
quelques fonctions

```
val ? : 'a tree -> int

val ? : 'a tree -> 'a option

val ? : 'a tree -> 'a tree
```

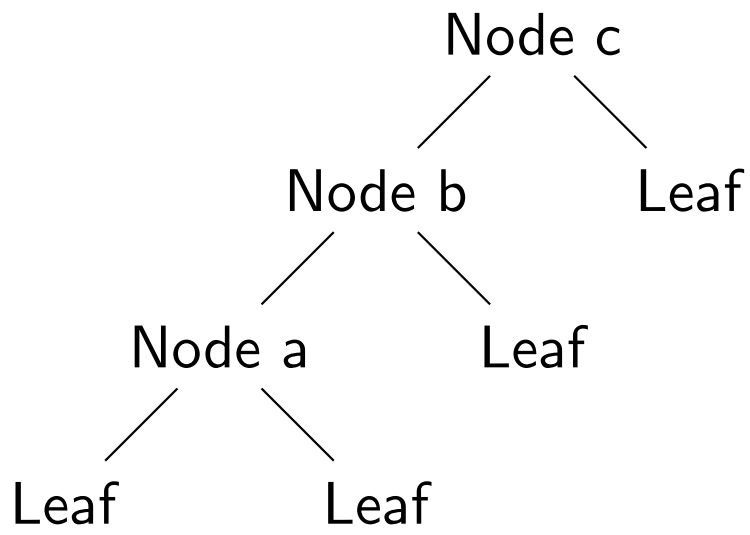
arbres : hauteur



$$
\begin{aligned}
&1 + \max \\
&\quad (1 + \max \\
&\qquad (1 + \max \\
&\qquad\quad 0 \\
&\qquad\quad 0) \\
&\qquad 0) \\
&\quad 0
\end{aligned}
$$

```
let rec depth : 'a tree -> int =
  function
  | Leaf -> 0
  | Node (l, _, r) -> 1 + max (depth l) (depth r)
```

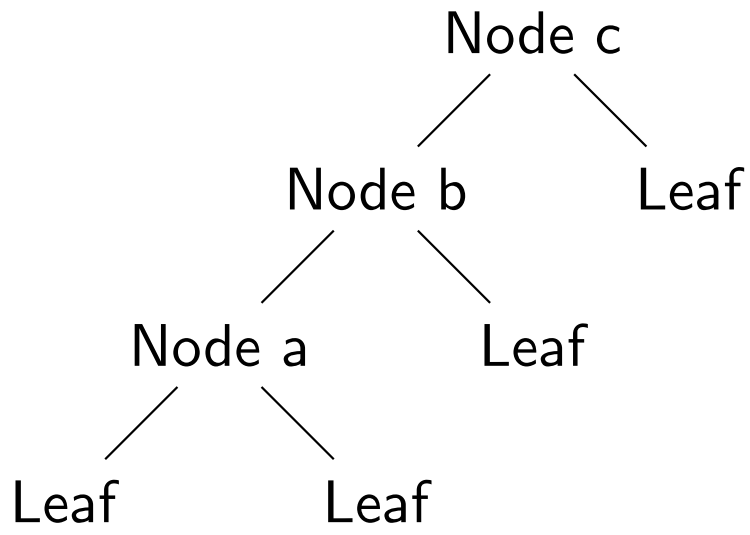
arbres : top

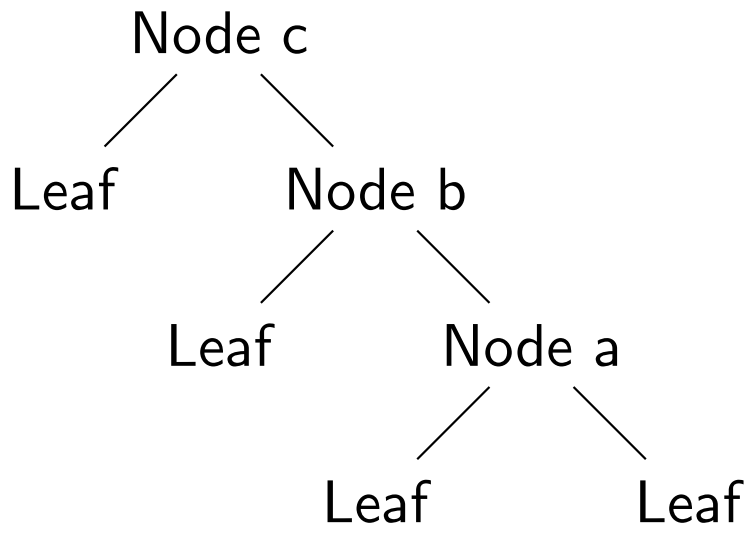


Some c

```
let top : 'a tree -> 'a option =
  function
  | Leaf -> None
  | Node (_, v, _) -> Some v
```

arbres : pivot



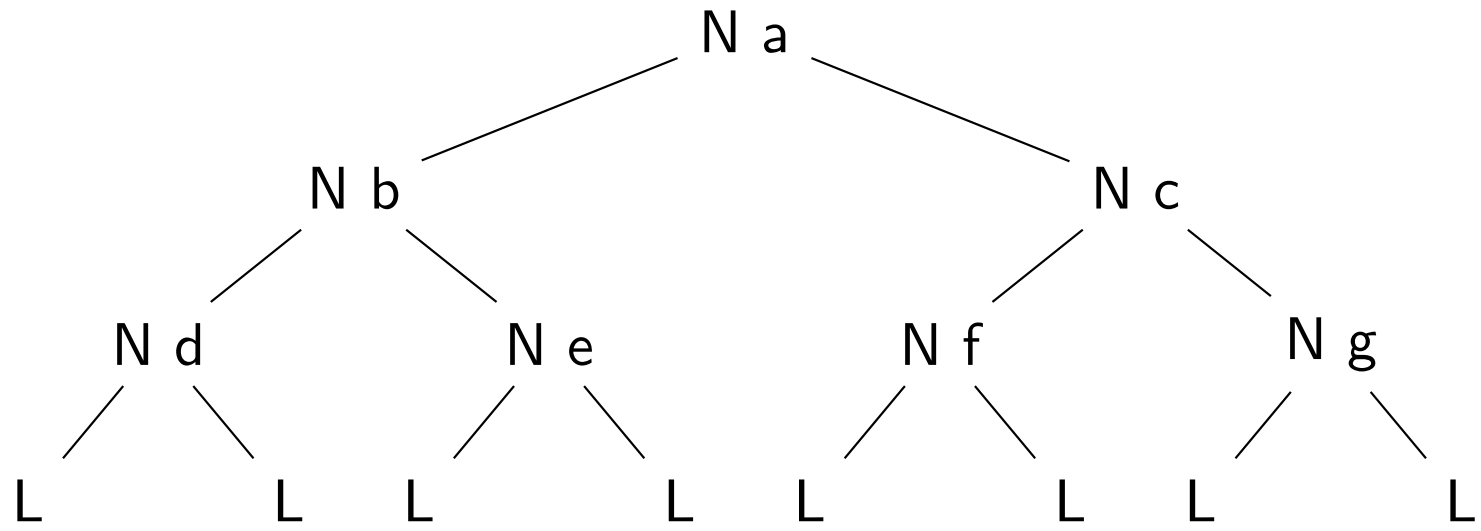



```
let rec pivot : 'a tree -> 'a tree =
  function
  | Leaf -> Leaf
  | Node (l, v, r) -> Node (pivot r, v, pivot l)
```

Arbres complets fortement typés

arbres complets

on s'intéresse maintenant à la construction d'arbres **complets**



peut-on donner une interface garantissant qu'on puisse construire *seulement* des arbres complets ?

arbres complets : avec types fantômes

c'est possible avec des types fantômes !

arbres complets : avec types fantômes

c'est possible avec des types fantômes !

astuce : entiers (unaires) "fortement typés" pour la hauteur

```
type z = Z
type 'a s = S of 'a
```

```
# let zero = Z;;
val zero : z = Z
# let three = S (S (S Z));;
val three : z s s s = S (S (S Z))
```

arbres complets : avec types fantômes

```
type ('a, 'd) ptree
(* type ('a, 'd) ptree = 'a tree *)
```

arbres complets : avec types fantômes

```
type ('a, 'd) ptree
(* type ('a, 'd) ptree = 'a tree *)

val leaf : ('a, z) ptree
val node :
  ('a, 'n) ptree * 'a * ('a, 'n) ptree ->
  ('a, 'n s) ptree
```

arbres complets : avec types fantômes

```
type ('a, 'd) ptree
(* type ('a, 'd) ptree = 'a tree *)

val leaf : ('a, z) ptree
val node :
  ('a, 'n) ptree * 'a * ('a, 'n) ptree ->
  ('a, 'n s) ptree
```

Limitations :

- pour l'utilisateur : pas de pattern matching (`ptree` est abstrait)
- pour l'implémentation : on ne bénéficie pas des types riches

types algébriques : limitations

Le mieux qu'on puisse écrire avec un type algébrique standard :

```
type ('a, 'd) tree =
  | Leaf
  | Node of ('a, 'd) tree * 'a * ('a, 'd) tree
```

Limitations :

- Tous les constructeurs produisent un `('a, 'd) tree`
- Les seuls variables de types permises sont `'a` et `'d`

types algébriques : limitations

Le mieux qu'on puisse écrire avec un type algébrique standard :

```
type ('a, 'd) tree =
  | Leaf
  | Node of ('a, 'd) tree * 'a * ('a, 'd) tree
```

Limitations :

- Tous les constructeurs produisent un `('a, 'd) tree`
- Les seuls variables de types permises sont `'a` et `'d`

Ce qu'on veut : des **constructeurs** qui ont le type :

```
Leaf : ('a, z) tree
Node : ('a, 'n) tree * 'a * ('a, 'n) tree -> ('a, 'n s) tree
```

types algébriques généralisés (GADTs)

```
type (_, _) gtree =
  | Leaf : ('a, z) gtree
  | Node : ('a, 'n) gtree * 'a * ('a, 'n) gtree ->
           ('a, 'n s) gtree
```

```
# let l = Leaf ;;
val l : ('a, z) gtree = Leaf
# let t = Node (Leaf, 12, Leaf);;
val t : (int, z s) gtree = Node (Leaf, 12, Leaf)
# let t' = Node (t, 0, Leaf);;
                      ^^^^
Error: This expression has type (int, z) gtree
       but an expression was expected of type (int, z s) gtree
       Type z is not compatible with type z s
```

Point syntaxe :

ADT :

```
type 'a t =
| Foo of u
```

GADT :

```
type _ t =
| Foo : u -> 'a t
```

```
type (_, _) gtree =
  | Leaf : ('a, z) gtree
  | Node
# let
val l
# let
val t
# let
Error: This expression has type (int, z) gtree
       but an expression was expected of type (int, z s) gtree
       Type z is not compatible with type z s
```

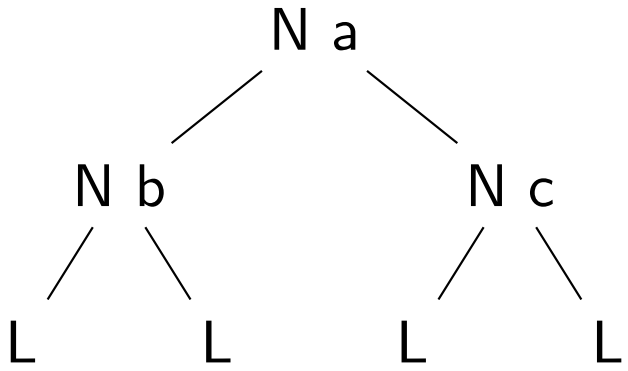
GADTs vs types fantômes

- Comme l'interface avec types fantômes, cette déclaration **restreint le type des valeurs construites en fonction du constructeur utilisé**

- En bonus, avec un GADT, les paramètres fantômes sont **directement liés au constructeurs** du type algébrique

- Ceci permet un **typage plus fin** dans les branches d'un **filtrage** ("pattern matching") sur un GADT

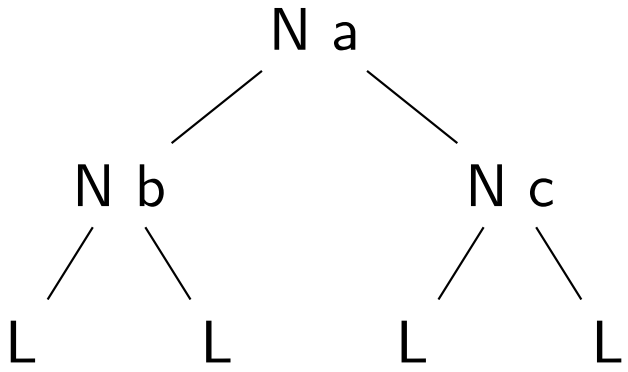
arbres complets (GADT) : depth



S (depth
S (depth
Z))

```
let rec depth : type a n. (a, n) gtree -> n =
  function ...
```

arbres complets (GADT) : depth



S (depth
S (depth
Z))

```
let rec depth : type a n. (a, n) gtree -> n =
  function
  | Leaf -> Z
  | Node (l, _, _) -> S (depth l)
```

pattern matching

Deux nouveaux phénomènes dans le cas d'un pattern matching :

- les types existants sont raffinés
- de nouveaux types sont introduits

```
type (_, _) gtree =
  | Leaf : ('a, z) gtree
  | Node : ('a, 'm) gtree * 'a * ('a, 'm) gtree -> ('a, 'm s) gtree

let rec depth : type a n. (a, n) gtree -> n =
  function
  | Leaf -> Z                    (* n = z *)
  | Node (l, _, _) ->            (* n = m s  (pour m frais) *)
      S (depth l)                (* l : (a, m) gtree        *)
                                 (* depth l : m             *)
```

annotations et récursion polymorphe

Attention : pour manipuler des GADTs, il faut généralement annoter les fonctions avec un type suffisamment polymorphe :

```
let rec depth : type a n. (a, n) gtree -> n =
  ...
  ... depth l ...      (* l : (a, m) gtree *)
```

ce style d'annotations est nécessaire pour permettre :

- le raffinement de types
- les appels récursifs à un type différent

pattern matching : cas impossibles

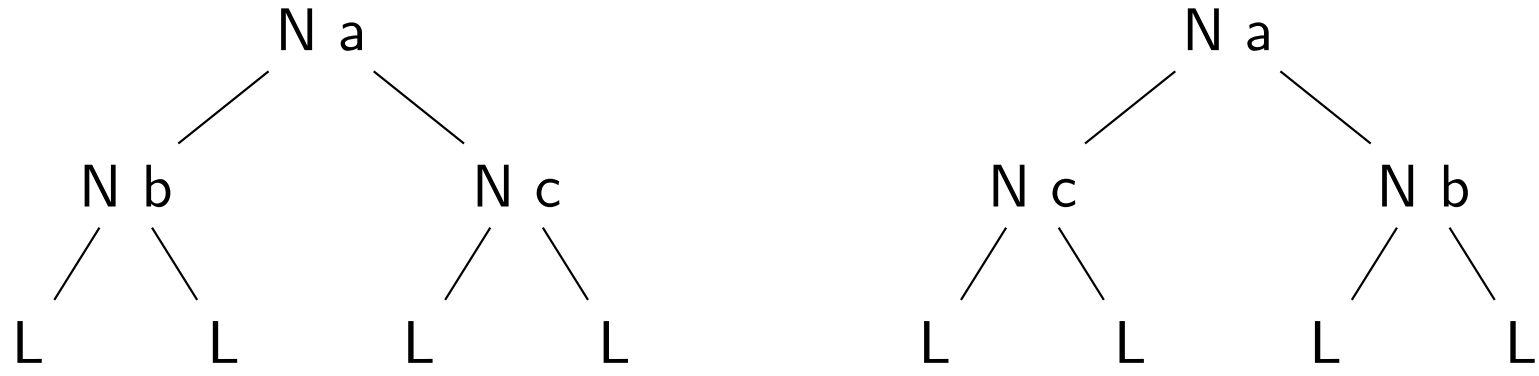
Le raffinement de type peut permettre d'éliminer des cas impossibles :

```
let top : type a n. (a, n s) gtree -> a =
  function
  | Node (_, v, _) -> v     (* Exhaustif ! *)
```

Dans la branche `Leaf`, on aurait `n s = z`.

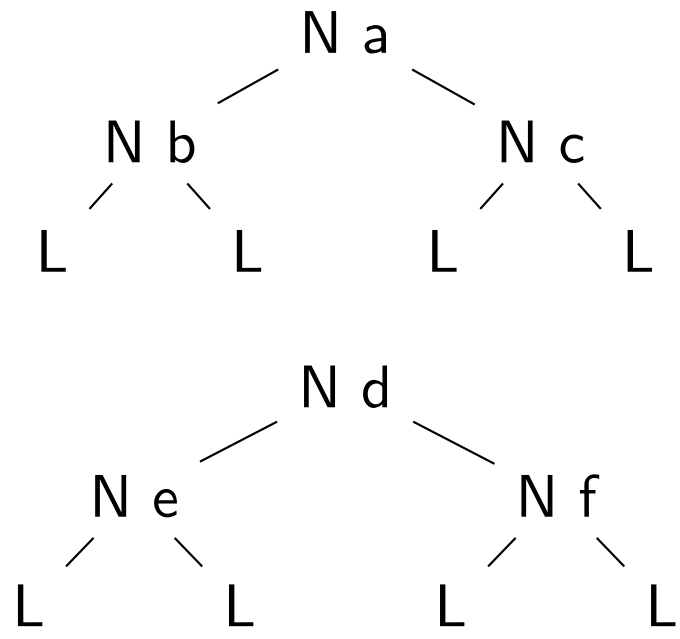
$\rightarrow$ impossible !

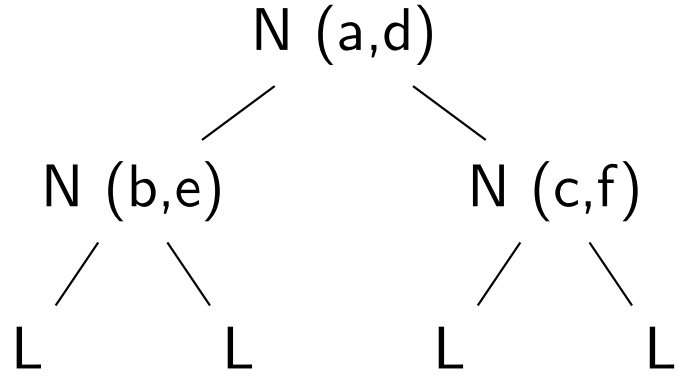
arbres complets (GADTs) : pivot




```
let rec pivot : type a n. (a, n) gtree -> (a, n) gtree =
  function
  | Leaf -> Leaf                          (* n = z *)
  | Node (l, v, r) ->                     (* n = m s  (pour m frais) *)
    Node (pivot r, v, pivot l)            (* l, r : (a, m) gtree     *)
                                          (* pivot l : (a, m) gtree  *)
```

arbres complets (GADTs) : zip






```
let rec zip :
  type a n. (a, n) gtree -> (a, n) gtree -> (a * a, n) gtree =
  fun x y -> match x, y with
    | Leaf, Leaf -> Leaf
    | Node (l, v, r), Node (l', v', r') ->
      Node (zip l l', (v, v'), zip r r')
```

arbres complets (GADTs) : zip

```
let rec zip :
  type a n. (a, n) gtree -> (a, n) gtree -> (a * a, n) gtree =
  fun x y -> match x, y with
    | Leaf, Leaf -> Leaf                          (* n = z  *)
    | Node (l, v, r), Node (l', v', r') ->  (* n = m s        *)
      Node (zip l l', (v, v'), zip r r')    (* (pour m frais) *)
```

Branche `Leaf`, `Node _` : `n = z` **et** `n = m s`

$\rightarrow$ impossible !

GADTs : désucrage

Comment interpréter la syntaxe GADT en terme d'opérations élémentaires sur les types ?

≪

```
type (_, _) gtree =
  | Leaf : ('a, z) gtree
  | Node : ('a, 'm) gtree * 'a * ('a, 'm) gtree -> ('a, 'm s) gtree
```

≫

= ? ?

GADTs, égalités et existentielles

Un GADT attache à chaque constructeur :

- des **égalités entre types**
- de nouvelles variables de type **quantifiés existentiellement**

```
type ('a, 'n) gtree =
  | Leaf : ['n = z] ('a, 'n) gtree
  | Node : [exists 'm]
           ['n = 'm s] ('a, 'm) gtree * 'a * ('a, 'm) gtree ->
                       ('a, 'n) gtree
```

La variable `'m` est ici **locale** au constructeur `Node`, et donc quantifiée existentiellement.

Égalités et existentielles (cas général)

Plus généralement :

```
type _ t =
  ...
  | Cstr : tyargs['a1..'an] -> tyret['a1..'an] t
```

doit être compris comme :

```
type 'a t =
  ...
  | Cstr of [exists 'a1..'an] tyargs['a1..'an]
            ['a = tyret['a1..'an]]
```

GADTs et égalités

En fait, on peut définir un GADT qui capture l'égalité entre deux types :

```
type ('a, 'b) eq = Equal : ('a, 'a) eq
(* c'est-à-dire, moralement : *)
type ('a, 'b) eq = Equal : ['a = 'b] ('a, 'b) eq
```

GADTs et égalités

En fait, on peut définir un GADT qui capture l'égalité entre deux types :

```
type ('a, 'b) eq = Equal : ('a, 'a) eq
(* c'est-à-dire, moralement : *)
type ('a, 'b) eq = Equal : ['a = 'b] ('a, 'b) eq


val refl : ('a, 'a) eq
val symm : ('a, 'b) eq -> ('b, 'a) eq
val trans : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq

val cast : ('a, 'b) eq -> 'a -> 'b
```

(défini dans la bibliothèque standard : `Type.eq`)

GADTs et égalités

On peut utiliser eq pour représenter les égalités attachées aux constructeurs :

```
type ('a, _) gtree =
  | Leaf : ('a, z) gtree
  | Node : ('a, 'n) gtree * 'a * ('a, 'n) gtree ->
           ('a, 'n s) gtree


type ('a, 'n) etree =
  | Leaf : ('n, z) eq -> ('a, 'n) etree
  | Node : ('n, 'm s) eq *
           ('a, 'm) etree * 'a * ('a, 'm) etree ->
           ('a, 'n) etree
```

GADTs et égalités

```
type ('a, _) gtree =
  | Leaf : ('a, z) gtree
  | Node : ('a, 'n) gtree * 'a * ('a, 'n) gtree ->
           ('a, 'n s) gtree
```

```
let rec depthG : type a n. (a, n) gtree -> n =
  function
  | Leaf -> Z                          (* n = z *)
  | Node (l, _, _) -> S (depthG l)     (* n = m s *)
```

GADTs et égalités

```
type ('a, 'n) etree =
  | Leaf : ('n, z) eq -> ('a, 'n) etree
  | Node : ('n, 'm s) eq *
           ('a, 'm) etree * 'a * ('a, 'm) etree ->
           ('a, 'n) etree
```

```
let rec depthE : type a n. (a, n) etree -> n =
  function
  | Leaf Equal -> Z                          (* n = z *)
  | Node (Equal, l, _, _) -> S (depthE l)    (* n = m s *)
```

GADTs et types existentiels

On ne peut pas définir un GADT “quantification existentielle” générique

Mais on peut le faire sur des cas concrets :

```
type elist =
  | EList : 'a list -> elist
```

`elist` correspond au type “$\exists\alpha, \alpha$ `list`”

le type des listes d'éléments d'un type inconnu

Application : tables associatives hétérogènes

Tables associatives standard

Structure de table associative `'a map`, dont les clefs sont des `string` et les valeurs des `'a` :

```
type 'a map

val empty : 'a map
val insert : 'a map -> string -> 'a -> 'a map
val find : 'a map -> string -> 'a option
```

Données hétérogènes

Peut-on avoir une map où des **clefs différentes** stockent des valeurs de **types différents** ?

Données hétérogènes

Peut-on avoir une map où des **clefs différentes** stockent des valeurs de **types différents** ?

```
type map
val empty : map
val insert : map -> string -> 'a -> map

val find : map -> string -> ???
```

Données hétérogènes

Peut-on avoir une map où des **clefs différentes** stockent des valeurs de **types différents** ?

```
type map
val empty : map
val insert : map -> string -> 'a -> map

val find : map -> string -> ???
```

(le problème est-il plus simple si les valeurs sont soit `int` ou `bool` ?)

Clefs hétérogènes

Idée : une **clef** indique le type de valeur correspondant !

Clefs hétérogènes

Idée : une **clef** indique le type de valeur correspondant !

```
type map
type 'a key  (* clef associée à une valeur de type 'a *)

val empty : map
val insert : map -> 'a key -> 'a -> map
val find : map -> 'a key -> 'a option
```

Clefs hétérogènes

Idée : une **clef** indique le type de valeur correspondant !

```
type map
type 'a key  (* clef associée à une valeur de type 'a *)

val empty : map
val insert : map -> 'a key -> 'a -> map
val find : map -> 'a key -> 'a option


val int_key : string -> int key
val bool_key : string -> bool key
```

Implémentation

Comment implémenter cette interface ?

$\rightarrow$ voir `assoc.ml`

Implémentation

Comment implémenter cette interface ?

→ voir `assoc.ml`

Illustre :

- type existentiel pour les paires clef/valeur de la liste
- type singleton (“`'a sort`”) pour représenter les types à runtime
- type égalité (`('a, 'b) eq`) pour test d'égalité fortement typé

Application : printf bien typé

```
# Printf.printf "abc";;
abc- : unit = ()
```

```
# Printf.printf "abc";;
abc- : unit = ()
# Printf.printf "abc %d";;
```

```
# Printf.printf "abc";;
abc- : unit = ()
# Printf.printf "abc %d";;
- : int -> unit = <fun>
```

```
# Printf.printf "abc";;
abc- : unit = ()
# Printf.printf "abc %d";;
- : int -> unit = <fun>
# Printf.printf "abc %d %s";;
```

```
# Printf.printf "abc";;
abc- : unit = ()
# Printf.printf "abc %d";;
- : int -> unit = <fun>
# Printf.printf "abc %d %s";;
- : int -> string -> unit = <fun>
```

???

```
# Printf.printf "abc";;
abc- : unit = ()
# Printf.printf "abc %d";;
- : int -> unit = <fun>
# Printf.printf "abc %d %s";;
- : int -> string -> unit = <fun>
```

???

le compilateur OCaml triche (un peu) :

```
# Printf.printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
```

Un argument donné à `printf` ressemble à une chaîne de caractères mais est parsé différemment

Réimplémentons printf

```
val printf : 'a fmt -> 'a
```

où `'a fmt` est une “chaîne de format” spécifiant ce que doit faire `printf`

De quelles manières veut-on pouvoir construire des `'a fmt` ?

$\longrightarrow$ `printf.ml`

GADTs : techniques essentielles

Type existentiel

Type existentiel :

lorsqu'un type est "caché" par un constructeur de GADT.

Exemple, avec `'n` **quantifié existentiellement** :

```
type 'a etree =
  | Etree : ('a, 'n) gtree -> 'a etree

let l : int etree list =
  [ Etree Leaf; Etree (Leaf, Node 1, Leaf) ]
```

Type singleton

Type singleton : type habité par une seule valeur

Sert souvent à fournir des *représentations de types à runtime*

Exemple :

```
type z = Z
type 'a s = S of 'a
let _ : z s s = S (S Z)
```

mais aussi :

```
type _ sort =
  | Int : int sort
  | String : string sort
```

Type égalité

Type égalité : type spécifiant l'égalité entre deux types

```
type (_, _) eq = Equal : ('a, 'a) eq
```

Utile notamment en combinaison avec les types singletons, pour tester (dynamiquement) l'égalité entre deux `sort` et renvoyer un témoin d'égalité entre les types OCaml correspondants

```
val sort_eq : 'a sort -> 'b sort -> ('a, 'b) eq option
(* au lieu de : *)
val sort_eq_weak : 'a sort -> 'b sort -> bool
```

Application avancée : données structurées fortement typées

types riches pour données structurées

Les GADTs permettent de refléter au niveau des types la structure de données structurées.

Exemple : (mini) JSON, décrit “simplement” avec un type algébrique

```
type ujson =
  | UNum : int -> ujson
  | UStr : string -> ujson
  | UBool : bool -> ujson
  | UArr : ujson list -> ujson

let j : ujson =
  UArr [ UStr "one"; UBool true; UNum 42 ]
```

types riches pour données structurées : JSON

Avec un GADT associant un type OCaml à chaque constructeur :

```
type _ tjson =
  | Num : int -> int tjson
  | Str : string -> string tjson
  | Bool : bool -> bool tjson
  | Arr : 'a tarr -> 'a tjson
and _ tarr =
  | Nil : unit tarr
  | (::) : 'a tjson * 'b tarr -> ('a * 'b) tarr
```

```
let _ = Arr (Str "one" :: Bool true :: Num 42 :: Nil)
- : (string * (bool * (int * unit))) tjson
```

types riches pour données structurées : JSON

```
(* Négation des booléens dans un JSON (simplement typé) *)
let rec unegate : ujson -> ujson = ...

(* Négation des booléens dans un JSON (richement typé) *)
let rec negate : type a. a tjson -> a tjson = function
  | Bool true -> Bool false
  | Bool false -> Bool true
  | Arr arr -> Arr (negate_arr arr)
  | v -> v
and negate_arr : type a. a tarr -> a tarr = function
  | Nil -> Nil
  | j :: js -> negate j :: negate_arr js
```

séparer types et données

Il est parfois utile de séparer la **description** du format de données, des données elles mêmes.

On peut ainsi définir le **schéma** richement typé d'une structure JSON :

```
type _ tyjson =
  | TyNum : int tyjson
  | TyStr : string tyjson
  | TyBool : bool tyjson
  | TyArr : 'a tyarr -> 'a tyjson
and _ tyarr =
  | TyNil : unit tyarr
  | (::) : 'a tyjson * 'b tyarr -> ('a * 'b) tyarr
```

un "`a tyjson`" ne contient pas de données, mais décrit la forme de valeurs de type `a` !

séparer types et données

```
(* avec tjson : description de type et données mélangées *)
let _ = Arr (Str "one" :: Bool true :: Num 42 :: Nil)
- : (string * (bool * (int * unit))) tjson

(* avec tyjson : description de type séparée des données *)
let _ = Arr (Str :: Bool :: Num :: Nil)
- : (string * (bool * (int * unit))) tyjson
let _ = ("one", (true, (42, ())))
- : (string * (bool * (int * unit)))
```

séparer types et données

```
let rec negate : type a. a tyjson -> a -> a =
  fun t v -> match t, v with
    | TyBool, true -> false
    | TyBool, false -> true
    | TyArr a, arr -> negate_arr a arr
    | _, v -> v
and negate_arr : type a. a tyarr -> a -> a =
  fun t v -> match t, v with
    | TyNil, () -> ()
    | j :: js, (a, b) -> (negate j a, negate_arr js b)
```

validation de données

On peut vérifier que des données simplement typées se conforment à un schéma donné, et renvoyer alors les *données brutes* :

```
let rec unpack_ujson : type a. a tyjson -> ujson -> a option =
  fun ty v -> match ty, v with
  | TyStr, UStr s -> Some s
  | TyNum, UNum u -> Some u
  | TyBool, UBool b -> Some b
  | TyArr a, UArr arr -> ...
  | _ -> None
```

Résumé

- les GADTs permettent d'annoter les constructeurs avec des types riches
- le pattern-matching raffine les types et révèle des égalités
- types existentiels pour cacher de l'information
- types singletons pour "calculer" sur les types ou les représenter

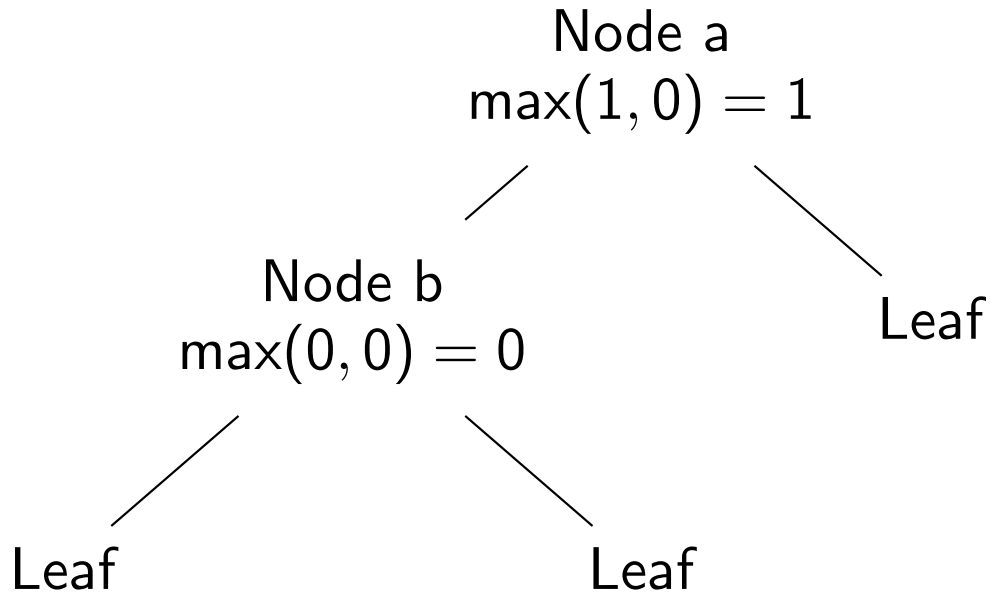
Quand utiliser les GADTs ?

- le typage fort peut renforcer les invariants de vos programmes et éviter les bugs
- l'interprétation de GADTs en terme de propositions logiques et preuves est parfois utile... mais pour écrire des preuves, on se tourne généralement plutôt vers Coq, Agda, etc.

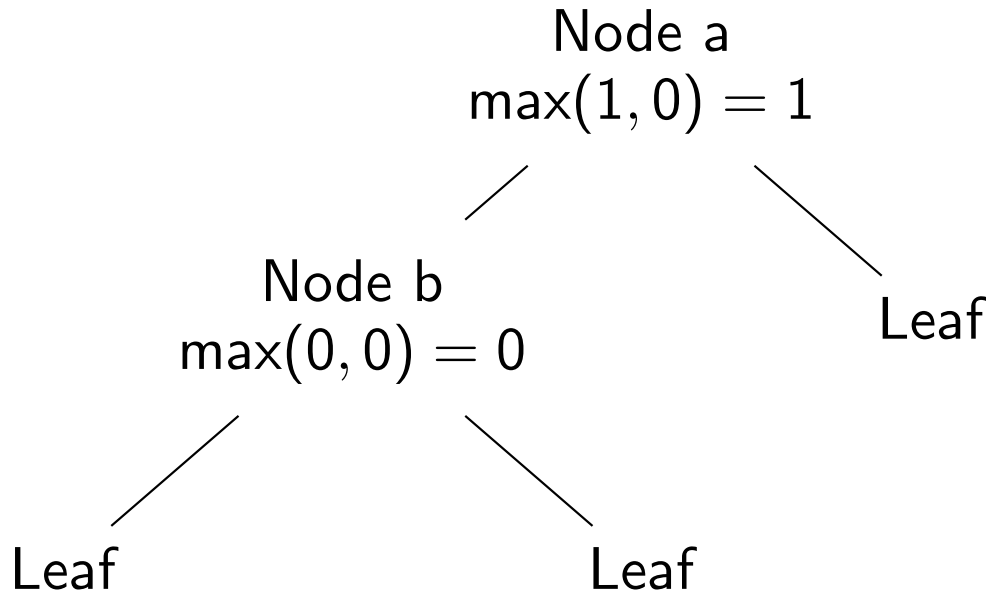
Arbres arbitraires annotés par leur hauteur

arbres annotés par leur hauteur

Les arbres complets ont une structure rigide. Mais il est également possible d'annoter des arbres *arbitraires* avec des types riches spécifiant leur hauteur.



arbres annotés par leur hauteur




```
type ('a, _) dtree =
  | LeafD : ('a, z) dtree
  | NodeD :
    ('a, 'm) dtree * 'a * ('a, 'n) dtree * ('m, 'n, 'o) max ->
    ('a, 'o s) dtree
```

le prédicat `max`

Idée : définir `max` comme un prédicat.

Le type "`(a, b, c) max`" correspond à la proposition $\max(a, b) = c$.
Une valeur de ce type correspond à une preuve de la proposition

On se donne les règles de raisonnement :

$$\text{MAX-EQ}\ \frac{}{\max(a, a) = a} \qquad \text{MAX-FLIP}\ \frac{\max(a, b) = c}{\max(b, a) = c} \qquad \text{MAX-SUC}\ \frac{\max(a, b) = a}{\max(a + 1, b) = a + 1}$$

`max` : définition

on définit `max` comme le GADT correspondant à ces règles de raisonnement :

```
type (_, _, _) max =
  | MaxEq : ('a, 'a, 'a) max
  | MaxFlip : ('a, 'b, 'c) max -> ('b, 'a, 'c) max
  | MaxSuc : ('a, 'b, 'a) max -> ('a s, 'b, 'a s) max
```

```
# MaxFlip (MaxSuc (MaxSuc (MaxEq (S Z))));;
- : (z s, z s s s, z s s s) max = ...
```

fonction `max`

étant donné une preuve de `(a, b, c) max` et `a` et `b`, on peut calculer `c` :

```
type (_, _, _) max =
  | MaxEq : ('a, 'a, 'a) max
  | MaxFlip : ('a, 'b, 'c) max -> ('b, 'a, 'c) max
  | MaxSuc : ('a, 'b, 'a) max -> ('a s, 'b, 'a s) max

let rec max : type a b c. (a, b, c) max -> a -> b -> c
= fun mx m n -> match mx, m with
  | MaxEq, _ -> m                          (* a = b, a = c, m : c *)
  | MaxFlip mx', _ -> max mx' n m
  | MaxSuc mx', S m' -> S (max mx' m' n)   (* a = d s, c = d s *)
                                           (* mx' = (d, b, d) max *)
                                           (* m' : d *)
                                           (* max mx' m' n : d *)
```

dtrees : hauteur

```
type ('a, _) dtree =
  | LeafD : ('a, z) dtree
  | NodeD :
    ('a, 'm) dtree * 'a * ('a, 'n) dtree * ('m, 'n, 'o) max ->
    ('a, 'o s) dtree
```

```
let rec depthD : type a n. (a, n) dtree -> n =
  function
  | LeafD -> Z                                  (* n = z *)
  | NodeD (l, _, r, mx) ->                      (* n = o s  (o,p,q frais) *)
    S (max mx (depthD l) (depthD r))            (* l : (a, p) dtree       *)
                                                (* r : (a, q) dtree       *)
                                                (* mx : (p, q, o) max     *)
```

dtrees : top

```
let topD : type a n. (a, n s) dtree -> a =
  function NodeD (_, v, _, _) -> v
```

dtrees : pivot

```
let pivotD : type a n. (a, n) dtree -> (a, n) dtree =
  function
  | LeafD -> LeafD
  | NodeD (l, v, r, mx) ->          (* mx : (p, q, o) mx *)
    NodeD (pivotD r, v, pivotD l,
           MaxFlip mx)              (* MaxFlip mx : (q, p, o) mx *)
```