Generalized algebraic data types

February 2018

Last time: types as propositions

$$\Gamma \vdash A$$

This time: GADTs

$$a \equiv b$$

Recap: Phantom types

```
module File : sig
  type 'a t
  type readonly
  type readwrite
  val open_readwrite : string -> readwrite t
  val open_readonly : string -> readonly t
  val read : 'a t -> string
  val write : readwrite t -> string -> unit
end = struct
  type 'a t = int
  ...
```

Recap: Phantom types

```
module File : sig
  type 'a t
  type readonly
  type readwrite
  val open_readwrite : string -> readwrite t
  val open_readonly : string -> readonly t
  val read : 'a t -> string
  val write : readwrite t -> string -> unit
end = struct
  type 'a t = int
  ...
```

equality hidden outside module

equality visible inside module

Recap: Phantom types

```
module File : sig
  type 'a t
  type readonly
  type readwrite
  val open_readwrite : string -> readwrite t
  val open_readonly : string -> readonly t
  val read : 'a t -> string
  val write : readwrite t -> string -> unit
end = struct
  type 'a t = int
  ...
```

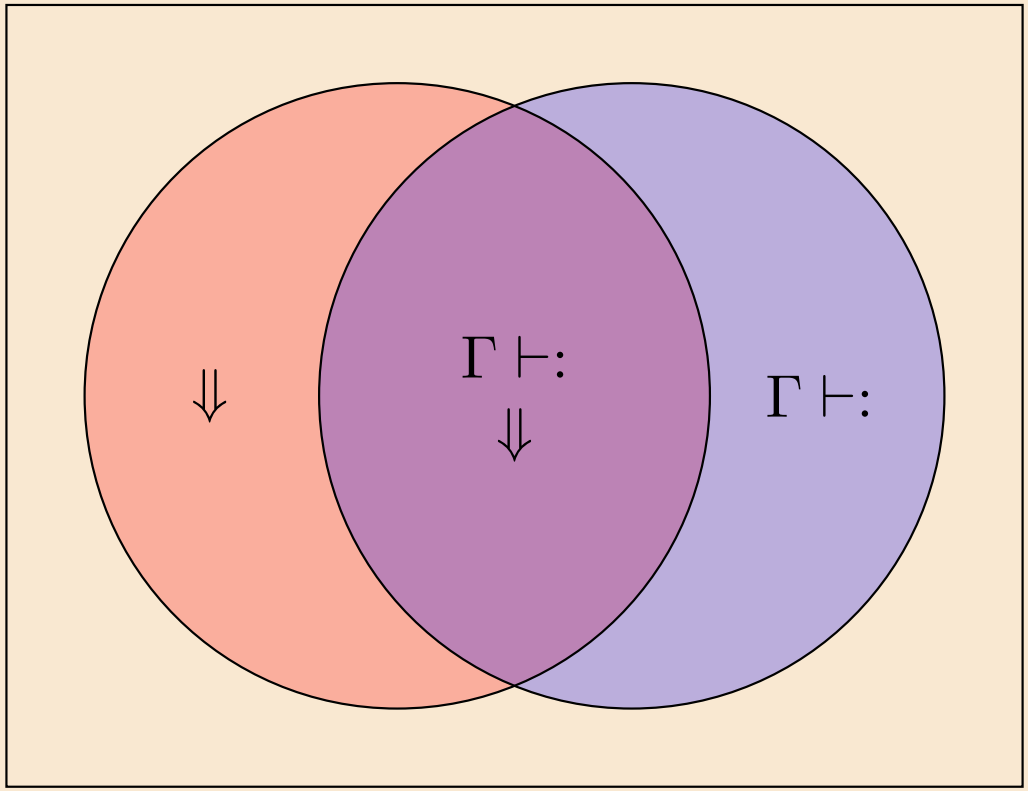
equality hidden outside module

equality visible inside module

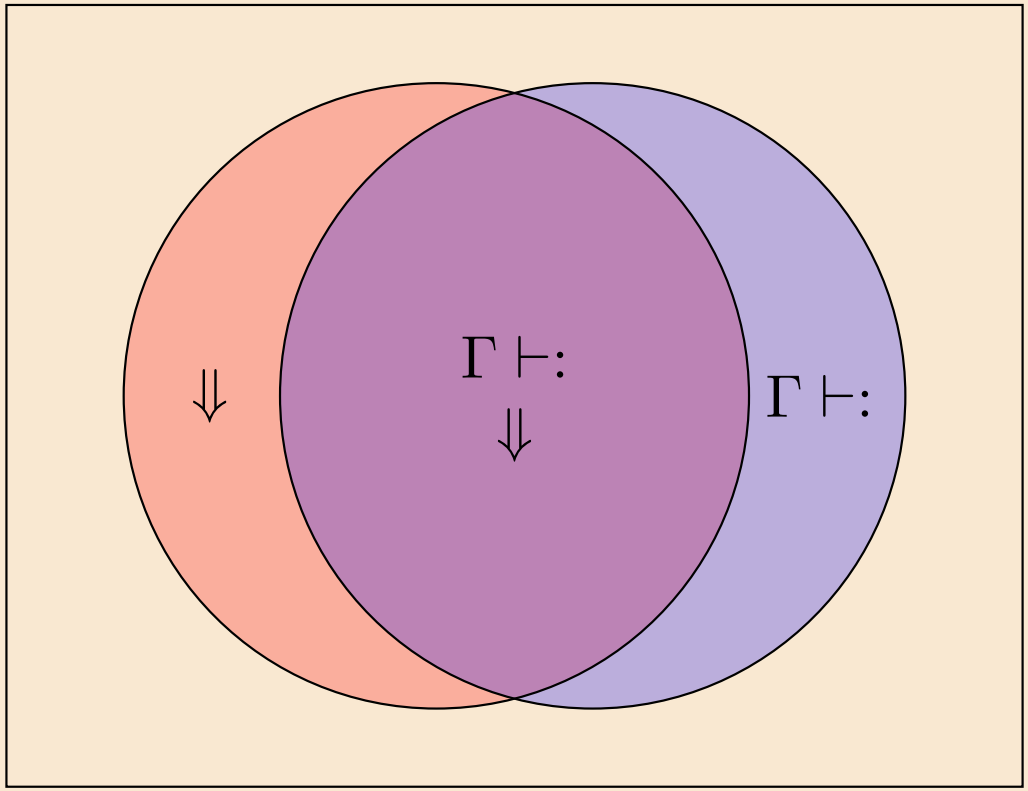
With **phantom types**:

- **safety outside** the `File` module (no writing to `readonly` files)
- **no safety inside** the `File` module (all operations available)

What we gain



What we gain



(Addtionally, some programs become faster!)

What it costs

We'll need to:

- **describe our data** more precisely
- strengthen the **relationship between data and types**
- look at programs through **a propositions-as-types lens**

What we'll write

Non-regularity in constructor return types

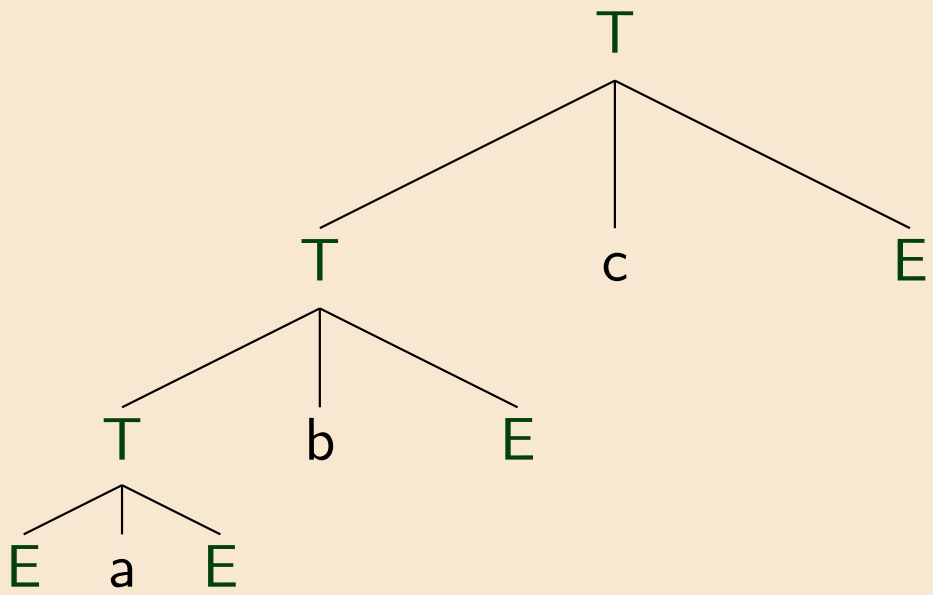
```
type _ t = T : t1 → t2 t
```

Locally abstract types:

```
let f : type a b. a t → b t = function ...

let g (type a) (type b) (x : a t) : b t = ...
```

Nested types

Unconstrained trees

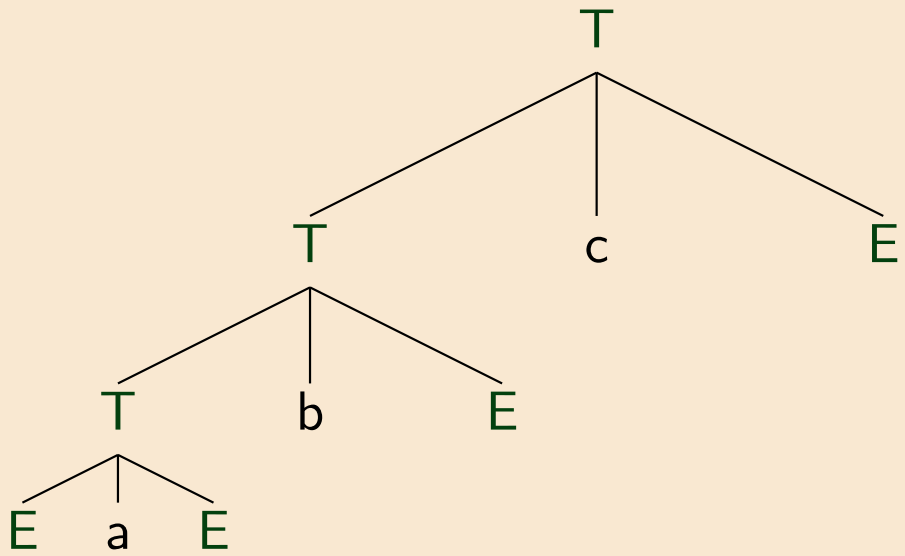



```
type 'a tree =
  Empty : 'a tree
| Tree : 'a tree * 'a * 'a tree → 'a tree
```

Term inference for unconstrained trees

```
val ? : 'a tree → int

val ? : 'a tree → 'a option

val ? : 'a tree → 'a tree
```

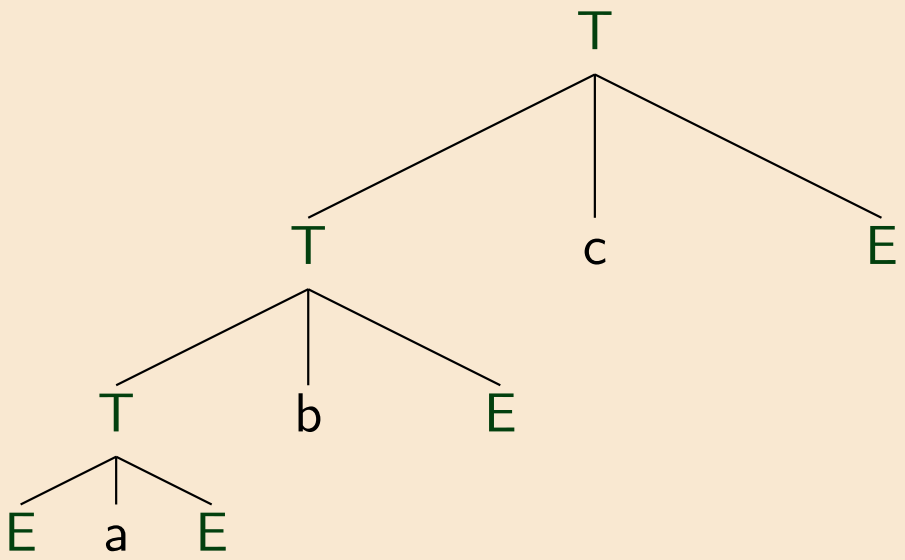
Unconstrained trees: depth




```
1 + max
      (1 + max
            (1 + max
                  0
                  0)
            0)
 0
```

```
let rec depth : 'a.'a tree → int =
 function
     Empty → 0
   | Tree (l,_,r) → 1 + max (depth l) (depth r)
```

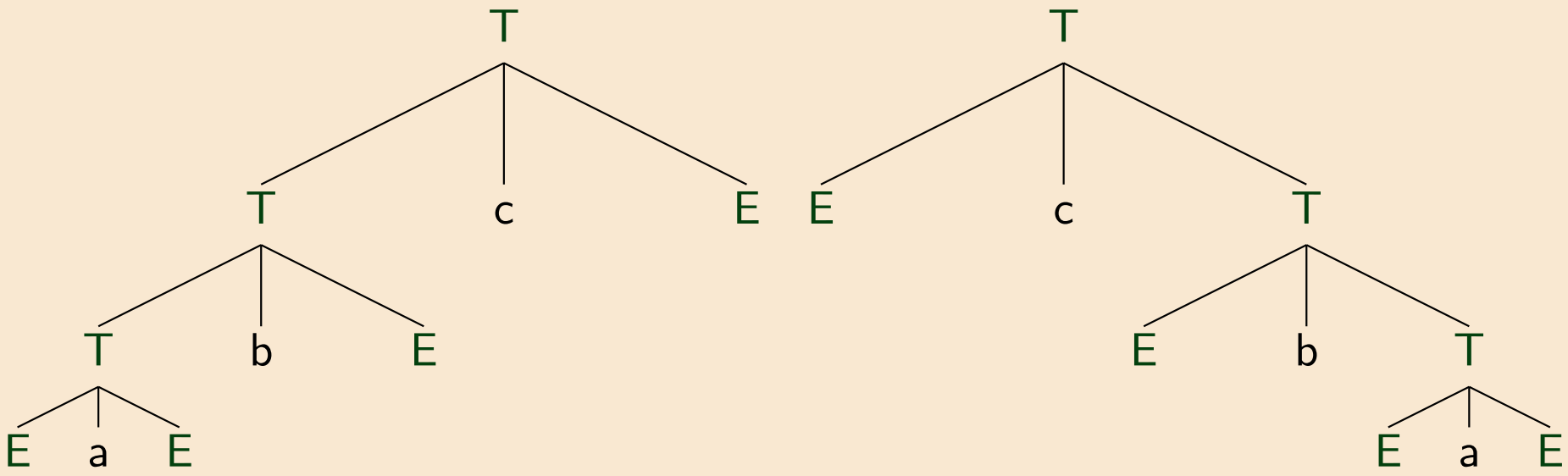
Unconstrained trees: top




```
Some c
```

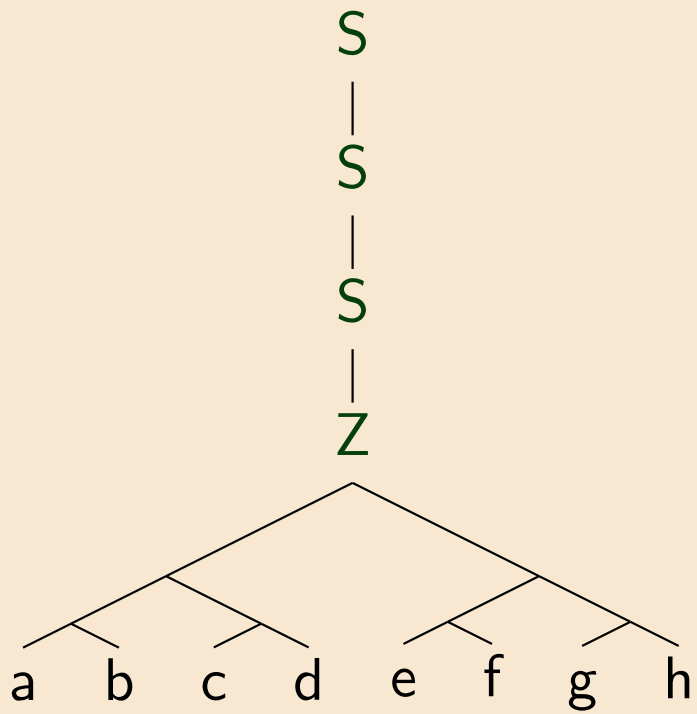
```
let top : 'a.'a tree → 'a option =
  function
    Empty → None
  | Tree (_,v,_) → Some v
```

Unconstrained trees: swivel



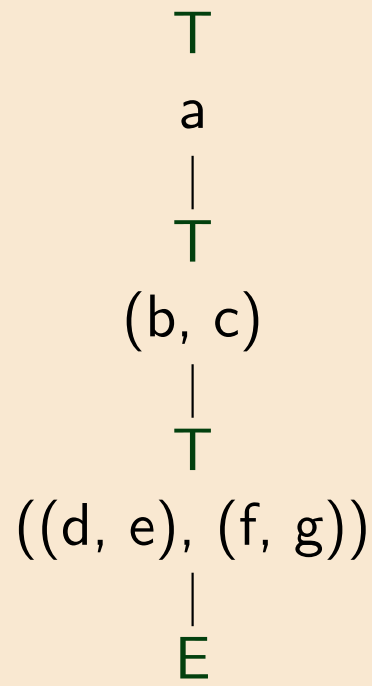

```
let rec swivel : 'a.'a tree → 'a tree =
 function
   Empty → Empty
 | Tree (l,v,r) → Tree (swivel r, v, swivel l)
```

Perfect leaf trees via nesting




```
type 'a perfect =
    ZeroP : 'a → 'a perfect
  | SuccP : ('a * 'a) perfect → 'a perfect
```

Perfect (branch) trees via nesting

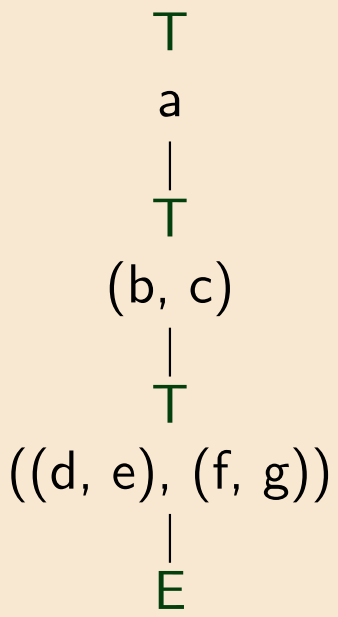



```
type _ ntree =
    EmptyN : 'a ntree
  | TreeN : 'a * ('a * 'a) ntree → 'a ntree
```

Term inference for perfect nested trees

```
val ? : 'a ntree → int

val ? : 'a ntree → 'a option

val ? : 'a ntree → 'a ntree
```

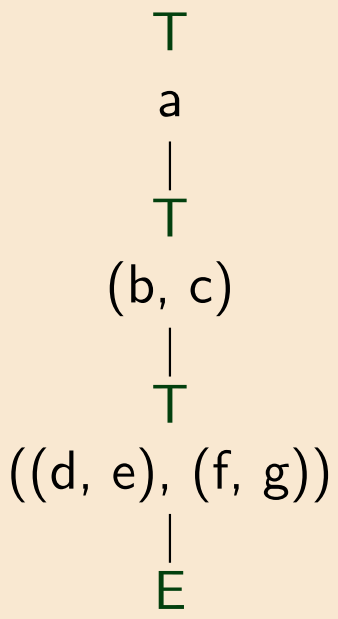
Perfect trees: depth




```
1 +
1 +
1 +
0
```

```
let rec depthN : 'a.'a ntree → int =
  function
      EmptyN → 0
    | TreeN (_,t) → 1 + depthN t
```

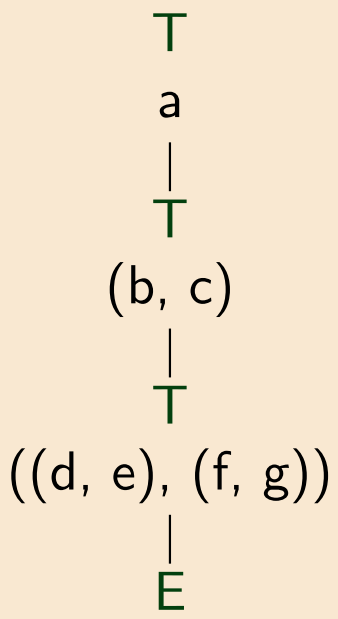
Perfect trees: top

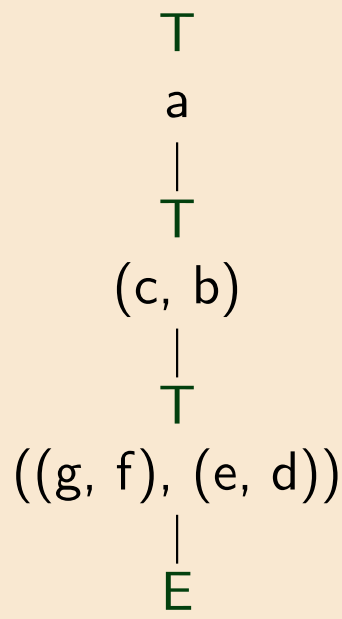



```
Some a
```

```
let rec topN : 'a.'a ntree → 'a option =
  function
    EmptyN → None
  | TreeN (v,_) → Some v
```

Perfect trees: swivel

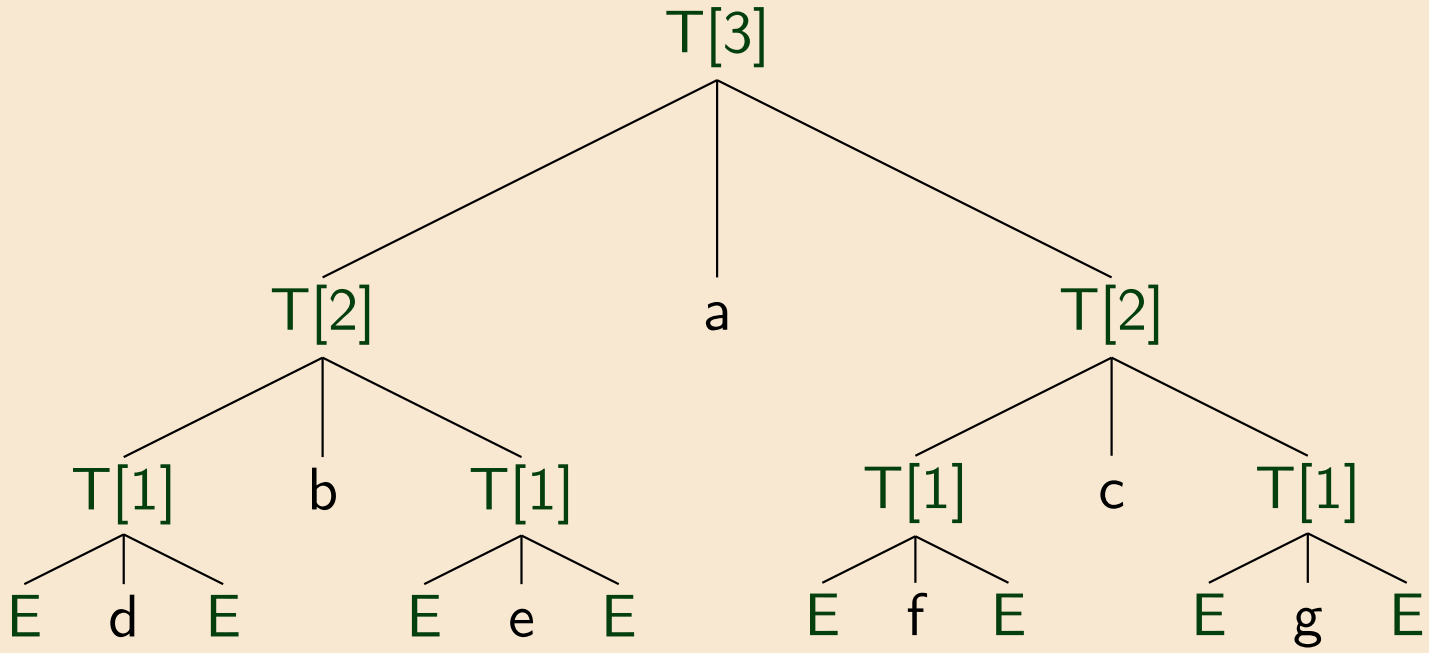





```
let rec swiv : 'a.('a→'a) → 'a ntree → 'a ntree =
  fun f t → match t with
    EmptyN → EmptyN
  | TreeN (v,t) →
    TreeN (f v,swiv (fun (x,y) → (f y, f x)) t)

let swivelN p = swiv id p
```

GADTs

Perfect trees, take two




```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree
```

Natural numbers

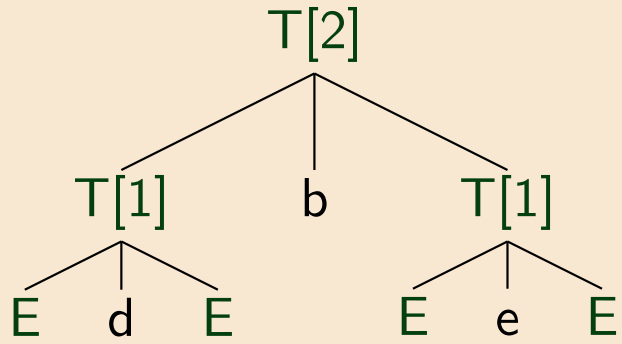
```
type z = Z
type _ s = S : 'n → 'n s
```

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

Term inference for perfect trees (GADTs)

```
val ? : ('a,'n) gtree → 'n

val ? : ('a,'n s) gtree → 'a

val ? : ('a,'n) gtree → ('a,'n) gtree
```

Perfect trees (GADTs): depth




```
S (depth
S (depth
    Z))
```

```
let rec depthG : type a n.(a, n) gtree → n =
  function
    EmptyG → Z
  | TreeG (l,_,_) → S (depthG l)
```

Perfect trees (GADTs): depth

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
        → ('a,'n s) gtree

let rec depthG : type a n.(a, n) gtree → n =
  function
    EmptyG → Z
  | TreeG (l,_,_) → S (depthG l)
```

Type refinement

In the `EmptyG` branch: `n ≡ z`

In the `TreeG` branch: `n ≡ m s` (for some `m`)

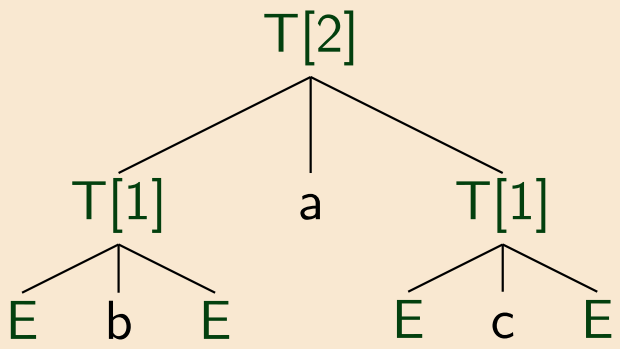
`l : (a, m) gtree`

`depthG l : m`

Polymorphic recursion

The argument to the recursive call has size `m` (s.t. `s m ≡ n`)

Perfect trees (GADTs): top



a

```
let topG : type a n.(a,n s) gtree → a =
  function TreeG (_,v,_) → v
```

Perfect trees (GADTs): top

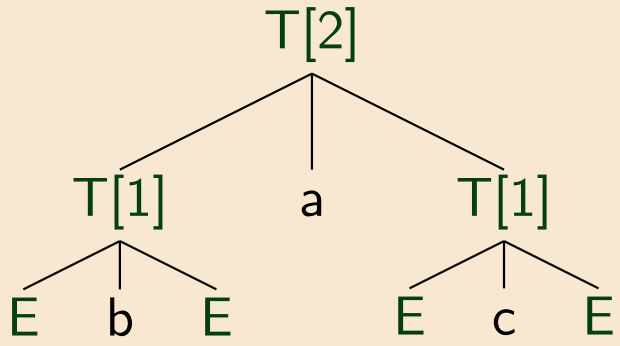
```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
        → ('a,'n s) gtree

let topG : type a n.(a,n s) gtree → a =
  function TreeG (_,v,_) → v
```

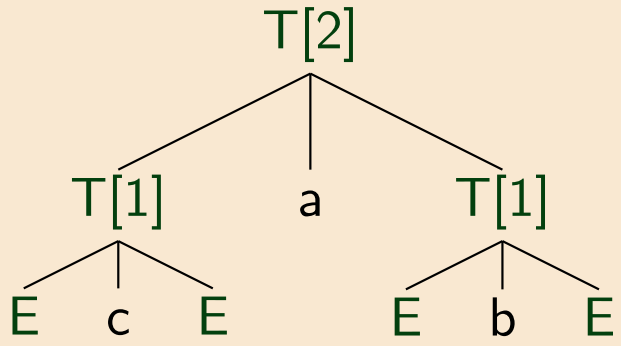
Type refinement

In an `EmptyG` branch we would have: `n s ≡ z`

— **impossible**!

Perfect trees (GADTs): swivel






```
let rec swivelG : type a n.(a,n) gtree → (a,n) gtree =
 function
    EmptyG → EmptyG
  | TreeG (l,v,r) → TreeG (swivelG r, v, swivelG l)
```

Perfect trees (GADTs): swivel

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree

let rec swivelG : type a n.(a,n) gtree → (a,n) gtree =
 function
     EmptyG → EmptyG
   | TreeG (l,v,r) → TreeG (swivelG r, v, swivelG l)
```

Type refinement

In the `EmptyG` branch: `n ≡ z`

In the `TreeG` branch: `n ≡ m s` (for some `m`)

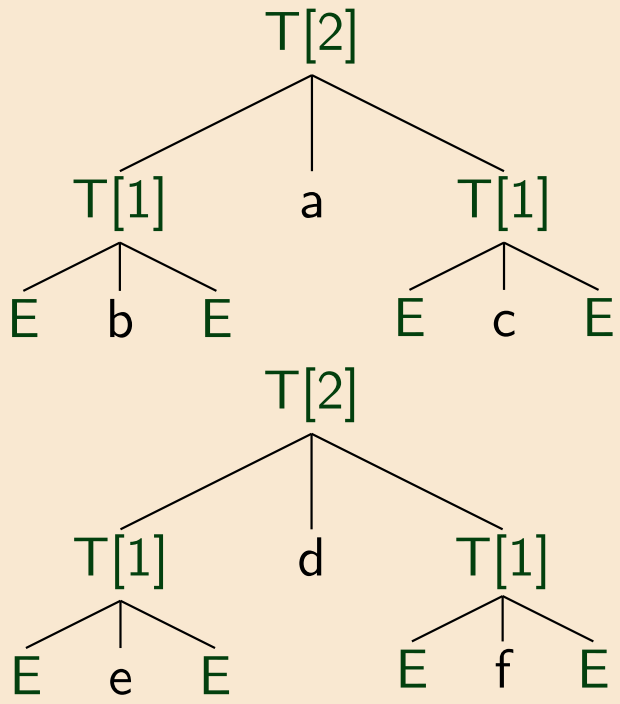
`l, r : (a, m) gtree`

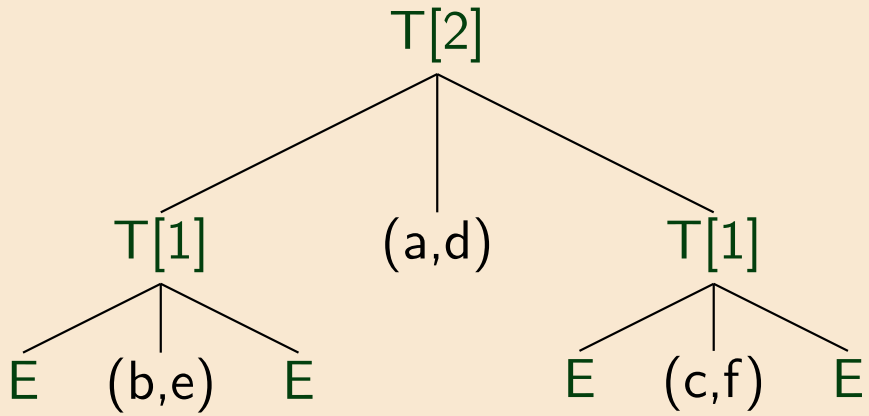
`swivelG l : (a, m) gtree`

Polymorphic recursion

The argument to the recursive call has size `m` (s.t. `s m ≡ n`)

Zipping perfect trees






```
let rec zipTree :
  type a b n.(a,n) gtree → (b,n) gtree →
                (a * b,n) gtree =
  fun x y → match x, y with
    EmptyG, EmptyG → EmptyG
  | TreeG (l,v,r), TreeG (m,w,s) →
    TreeG (zipTree l m, (v,w), zipTree r s)
```

Zipping perfect trees

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree

let rec zipTree :
  type a b n.(a,n) gtree → (b,n) gtree →
                (a * b,n) gtree =
  fun x y → match x, y with
    EmptyG, EmptyG → EmptyG
  | TreeG (l,v,r), TreeG (m,w,s) →
    TreeG (zipTree l m, (v,w), zipTree r s)
```

Type refinement

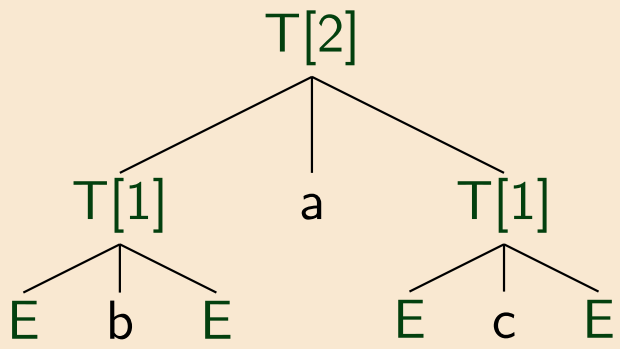
In the `EmptyG` branch: $\mathtt{n} \equiv \mathtt{z}$

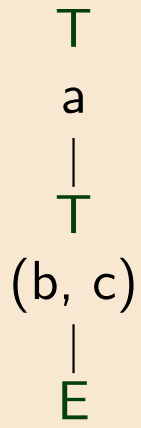
In the `TreeG` branch: $\mathtt{n} \equiv \mathtt{m\ s}$ (for some `m`)

`EmptyG, TreeG _` produces $\mathtt{n} \equiv \mathtt{z}$ **and** $\mathtt{n} \equiv \mathtt{m\ s}$

— **impossible**!

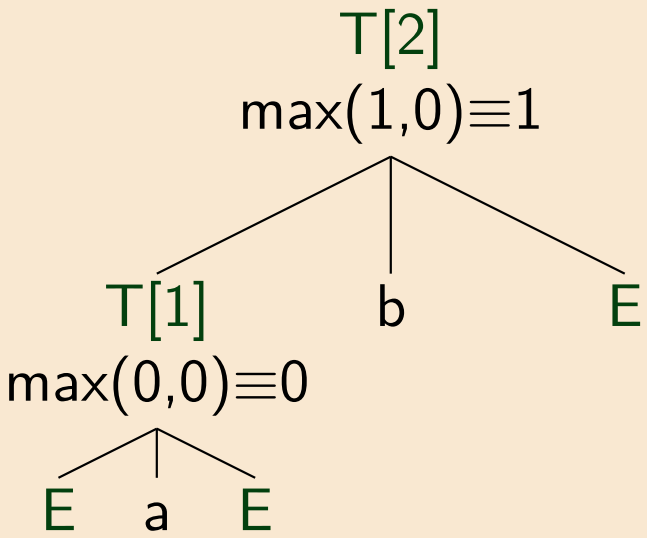
Conversions between perfect tree representations






```
let rec nestify : type a n.(a,n) gtree → a ntree =
  function
    EmptyG → EmptyN
  | TreeG (l, v, r) →
    TreeN (v, nestify (zipTree l r))
```

Depth-annotated trees




```
type ('a,_) dtree =
  EmptyD : ('a,z) dtree
| TreeD : ('a,'m) dtree * 'a * ('a,'n) dtree * ('m,'n,'o) max
        → ('a,'o s) dtree
```

The untyped maximum function

```
val max : 'a → 'a → 'a
```

Parametricity tells us that `max` is one of

```
fun x _ → x
fun _ y → y
```

A typed maximum function

```
val max : ('a,'b,'c) max → 'a → 'b → 'c
```

```
(max (a,b) ≡ c) → a → b → c
```

A typed maximum function: a max predicate

```
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
```

$$
\begin{array}{rcl}
\mathtt{a} \equiv \mathtt{b} & \rightarrow & \mathtt{max(a,b)} \equiv \mathtt{a} \\
\mathtt{max(a,b)} \equiv \mathtt{c} & \rightarrow & \mathtt{max(b,a)} \equiv \mathtt{c} \\
\mathtt{max(a,b)} \equiv \mathtt{a} & \rightarrow & \mathtt{max(a+1,b)} \equiv \mathtt{a+1}
\end{array}
$$

A typed maximum function

```
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
    | MaxFlip mx', _     → max mx' n m
    | MaxSuc mx' , S m' → S (max mx' m' n)
```

A typed maximum function

```
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
    | MaxFlip mx', _     → max mx' n m
    | MaxSuc mx' , S m' → S (max mx' m' n)
```

Type refinement

In the `MaxEq` branch: $\texttt{a} \equiv \texttt{b}$, $\texttt{a} \equiv \texttt{c}$

`m : c`

In the `MaxFlip` branch: *no refinement*

In the `MaxSuc` branch: $\texttt{a} \equiv \texttt{d s}$, $\texttt{c} \equiv \texttt{d s}$ (for some `d`)

`mx' : (d, b, d) max`

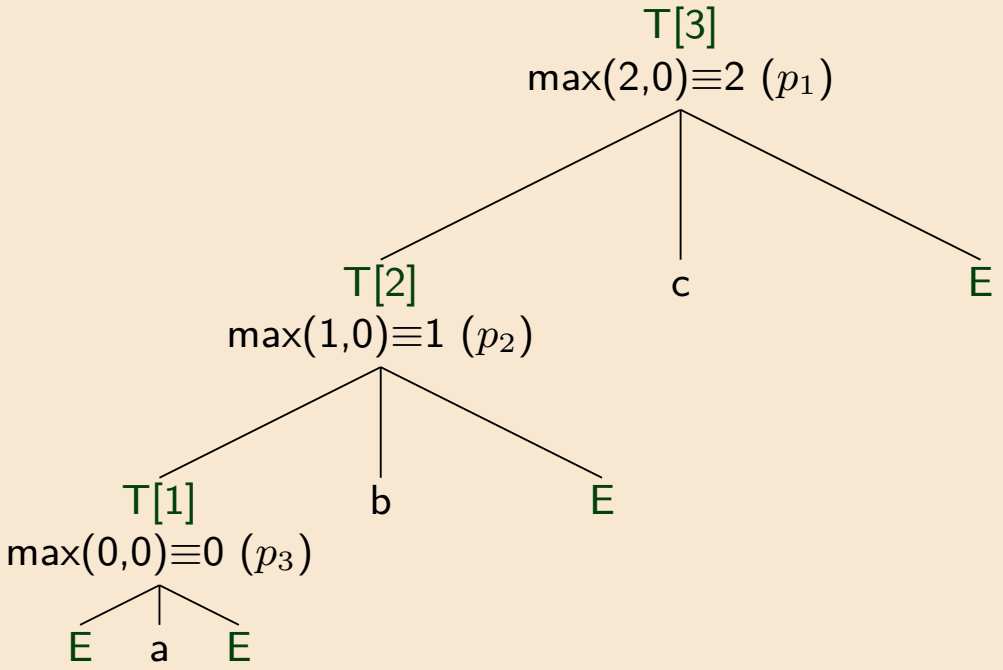
`m' : d`

`max mx' m' n : d`

Term inference for depth-annotated trees

```
val ? : ('a,'n) dtree → 'n

val ? : ('a,'n s) dtree → 'a

val ? : ('a,'n) dtree → ('a,'n) dtree
```

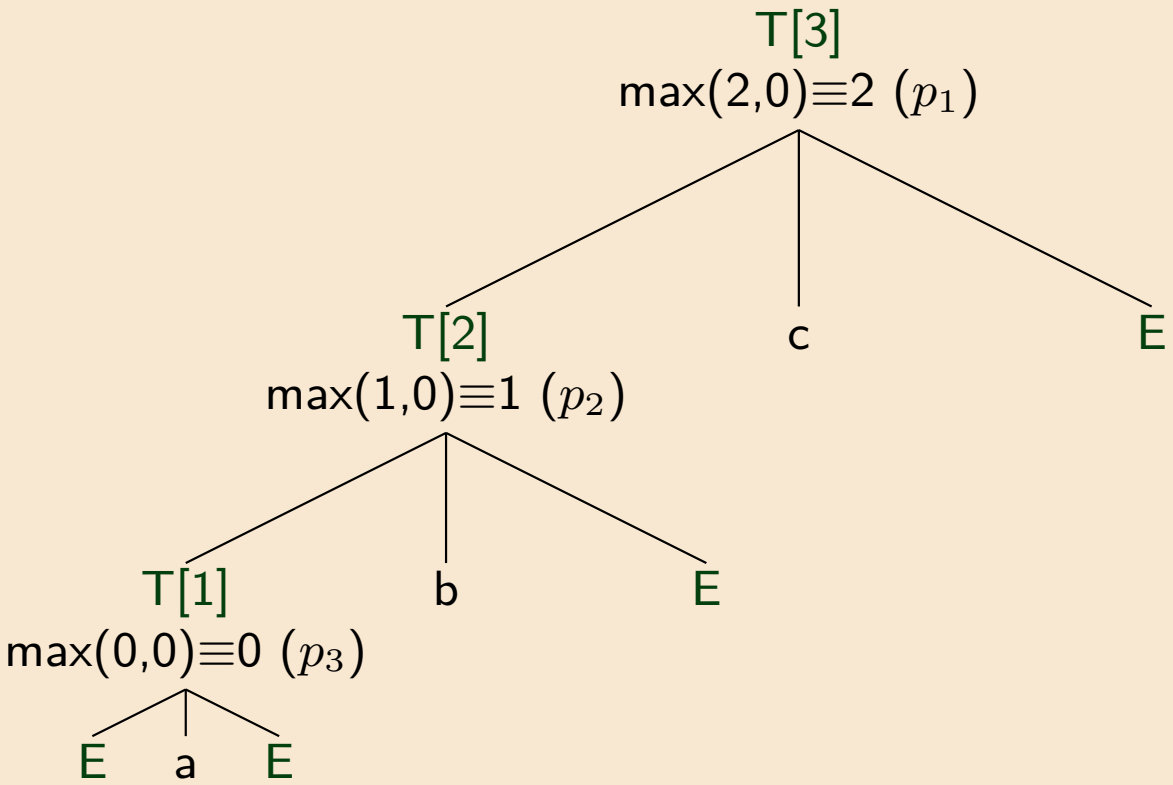
Depth-annotated trees: depth




```
S (max p1
   (S (max p2
       (S (max p3
           Z
           Z))
        Z))
   Z)
```

```
let rec depthD : type a n.(a,n) dtree → n =
 function
   EmptyD → Z
| TreeD (l,_,r,mx) → S (max mx (depthD l) (depthD r))
```

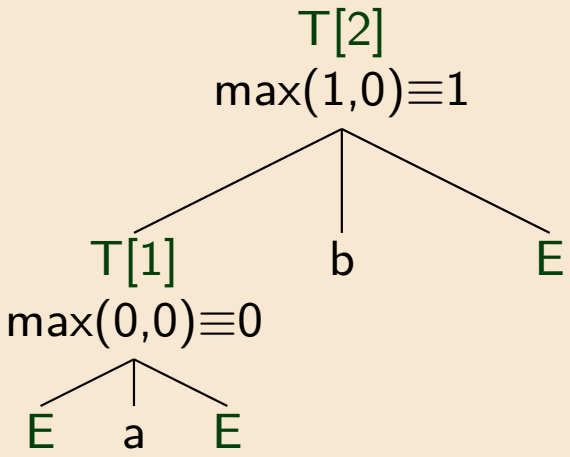
Depth-annotated trees: top

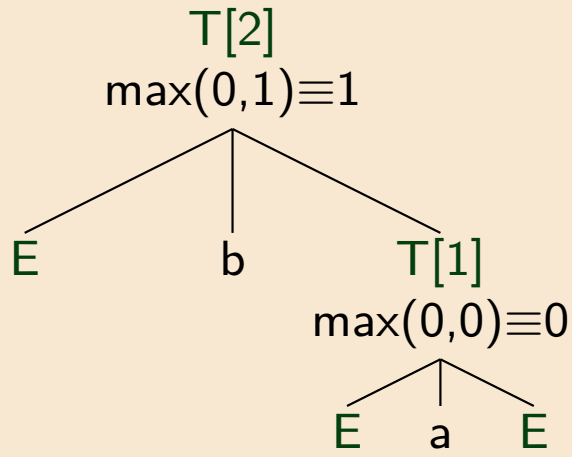


c

```
let topD : type a n.(a,n s) dtree → a =
  function TreeD (_,v,_,_) → v
```

Depth-annotated trees: swivel






```
let rec swivelD :
  type a n.(a,n) dtree → (a,n) dtree =
  function
    EmptyD → EmptyD
  | TreeD (l,v,r,m) →
     TreeD (swivelD r, v, swivelD l, MaxFlip m)
```

Efficiency

Efficiency: missing branches

```
let top : 'a.'a tree → 'a option = function
    Empty → None
  | Tree (_,v,_) → Some v
```

```
(function p                        (* ocaml -dlambda *)
  (if p
     (makeblock 0 (field 1 p))
     0a))
```

```
let topG : type a n.(a,n s) gtree → a = function
    TreeG (_,v,_) → v
```

```
(function p                        (* ocaml -dlambda *)
  (field 1 p))
```

Efficiency: zips

```
let rec zipTree :
  type a b n.(a,n) gtree → (b,n) gtree →
                (a * b,n) gtree =
  fun x y → match x, y with
    EmptyG, EmptyG → EmptyG
  | TreeG (l,v,r), TreeG (m,w,s) →
    TreeG (zipTree l m, (v,w), zipTree r s)
```

```
(letrec                          (* ocaml -dlambda *)
  (zipTree
     (function x y
       (if x
         (makeblock 0
           (apply zipTree (field 0 x) (field 0 y))
           (makeblock 0 (field 1 x) (field 1 y))
           (apply zipTree (field 2 x) (field 2 y)))
         0a)))
  (apply (field 1 (global Toploop!)) "zipTree" zipTree))
```

Equality

Recall: equality in System Fω

$$\texttt{Eql} = \lambda\alpha.\lambda\beta.\forall\phi::* \Rightarrow *.\phi\ \alpha \to \phi\ \beta$$

$$\texttt{refl} : \forall\alpha.\texttt{Eql}\ \alpha\ \alpha$$
$$\texttt{refl} = \Lambda\alpha\,.\,\Lambda\phi::* \Rightarrow *\,.\,\lambda\texttt{x}{:}\phi\ \alpha\,.\,\texttt{x}$$

$$\texttt{symm} : \forall\alpha.\forall\beta.\texttt{Eql}\ \alpha\ \beta \to \texttt{Eql}\ \beta\ \alpha$$
$$\texttt{symm} = \Lambda\alpha\,.\,\Lambda\beta\,.$$
$$\lambda\texttt{e}{:}(\forall\phi::* \Rightarrow *.\phi\ \alpha\ \to\ \phi\ \beta)\,.\,\texttt{e}\ [\lambda\gamma.\texttt{Eql}\ \gamma\ \alpha]\ (\texttt{refl}\ [\alpha])$$

$$\texttt{trans} : \forall\alpha.\forall\beta.\forall\gamma.\texttt{Eql}\ \alpha\ \beta \to \texttt{Eql}\ \beta\ \gamma \to \texttt{Eql}\ \alpha\ \gamma$$
$$\texttt{trans} = \Lambda\alpha\,.\,\Lambda\beta\,.\,\Lambda\gamma\,.$$
$$\lambda\texttt{ab}{:}\texttt{Eql}\ \alpha\ \beta\,.\,\lambda\texttt{bc}{:}\texttt{Eql}\ \beta\ \gamma\,.\,\texttt{bc}\ \ [\texttt{Eql}\ \alpha]\ \ \texttt{ab}$$

Equlity with GADTs

```
type (_, _) eql = Refl : ('a,'a) eql

val refl : ('a,'a) eql
val symm : ('a,'b) eql → ('b,'a) eql
val trans : ('a,'b) eql → ('b,'c) eql → ('a,'c) eql
```

```
module Lift (T : sig type _ t end) :
sig
  val lift : ('a,'b) eql → ('a T.t,'b T.t) eql
end

val cast : ('a,'b) eql → 'a → 'b
```

Building GADTs from algebraic types and equality

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree
```

```
type ('a,'n) etree =
   EmptyE : (z,'n) eql → ('a,'n) etree
| TreeE : ('n,'m s) eql *
 ('a,'m) etree * 'a * ('a,'m) etree → ('a,'n) etree
```

Building GADTs from algebraic types and equality

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree


let rec depthG : type a n.(a, n) gtree → n =
  function
    EmptyG → Z
  | TreeG (l,_,_) → S (depthG l)
```

Building GADTs from algebraic types and equality

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree
```

```
let rec depthG : type a n.(a, n) gtree → n =
  function
    EmptyG → Z                                  (* n = z *)
  | TreeG (l,_,_) → S (depthG l)
```

Building GADTs from algebraic types and equality

```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree
      → ('a,'n s) gtree
```

```
let rec depthG : type a n.(a, n) gtree → n =
  function
    EmptyG → Z                              (* n = z *)
  | TreeG (l,_,_) → S (depthG l)            (* n = m s *)
```

Building GADTs from algebraic types and equality

```
type ('a,'n) etree =
  EmptyE : ('n,z) eql → ('a,'n) etree
| TreeE : ('n,'m s) eql *
 ('a,'m) etree * 'a * ('a,'m) etree → ('a,'n) etree


let rec depthE : type a n.(a, n) etree → n =
  function
    EmptyE Refl → Z
  | TreeE (Refl, l,_,_) → S (depthE l)
```

Building GADTs from algebraic types and equality

```
type ('a,'n) etree =
  EmptyE : ('n,z) eql → ('a,'n) etree
| TreeE : ('n,'m s) eql *
 ('a,'m) etree * 'a * ('a,'m) etree → ('a,'n) etree


let rec depthE : type a n.(a, n) etree → n =
  function
    EmptyE Refl → Z                          (* n = z *)
  | TreeE (Refl, l,_,_) → S (depthE l)
```

Building GADTs from algebraic types and equality

```
type ('a,'n) etree =
  EmptyE : ('n,z) eql → ('a,'n) etree
| TreeE : ('n,'m s) eql *
 ('a,'m) etree * 'a * ('a,'m) etree → ('a,'n) etree


let rec depthE : type a n.(a, n) etree → n =
  function
    EmptyE Refl → Z                              (* n = z *)
  | TreeE (Refl, l,_,_) → S (depthE l)   (* n = m s *)
```

Reorientation

Where are we?

We've enriched data types with **indexes** describing the data.

Families of types: a type for each nat, each tree depth, etc.

Descriptive data types lead to **useful function types**.

Phantom types protect **abstractions** against misuse.
GADTs also protect **definitions**.

Compilers use the extra information to generate **better code**.

What's going on?

GADTs are about **type equalities** (and sometimes inequalities).

GADTs reveal things about **types** when you examine **data**.

We need machinery from earlier lectures:
existentials, polymorphic recursion, non-regularity.

GADTs lead to rich types that can be viewed as **propositions**.

Setting up types carefully leads to simple and **fast code**.

~~Generalized~~ algebraic data types