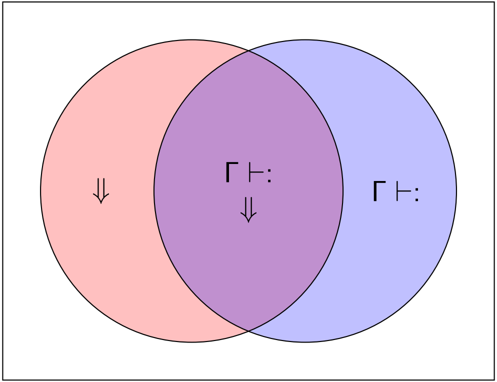
Last time: phantom types

```
type 'a t = int
```

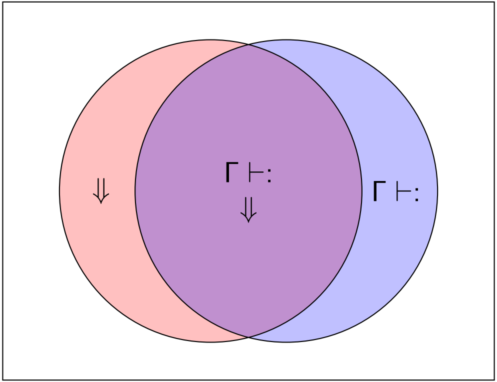
This time: GADTs

$$a \equiv b$$

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

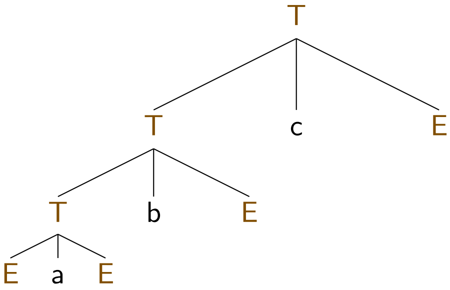
```
type _ t = T : t1 → t2 t
```

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 review

Unconstrained trees

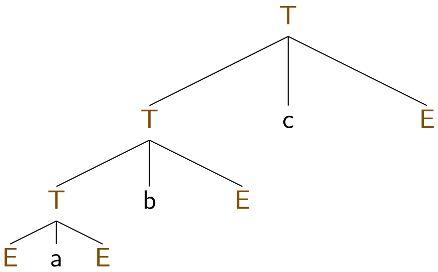



```
type 'a tree =
  Empty : 'a tree
| Tree : 'a tree * 'a * 'a tree → 'a tree
```

Functions on unconstrained trees

```
val ? : 'a tree → int

val ? : 'a tree → 'a option

val ? : 'a tree → 'a tree
```

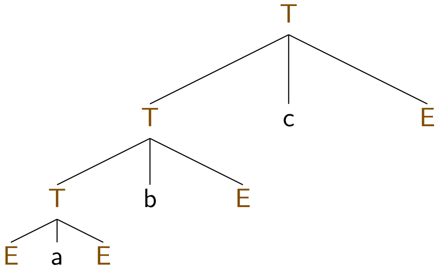
Unconstrained trees: depth




```
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)
```

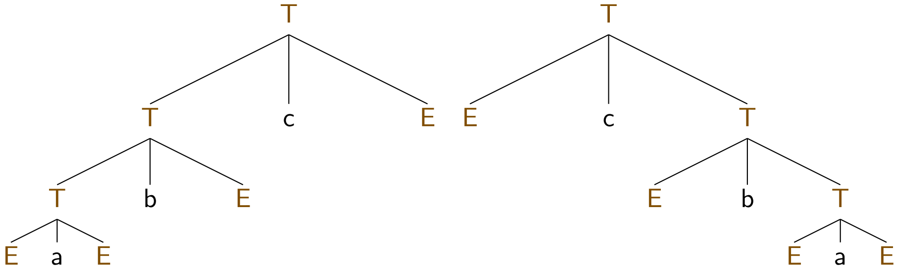
Unconstrained trees: top



Some c

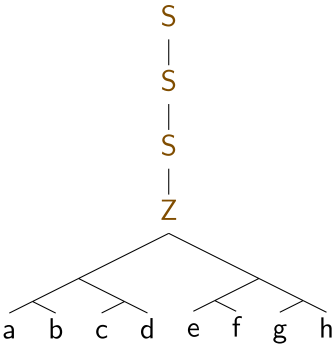
```
let top : 'a.'a tree → 'a option =
  function
    Empty → None
  | Tree (_,v,_) → Some v
```

Unconstrained trees: swivel



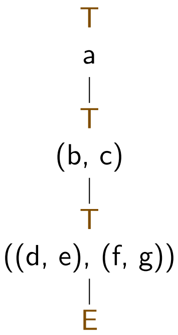

```
let rec swivel : 'a.'a tree → 'a tree =
  function
    Empty → Empty
  | Tree (l,v,r) → Tree (swivel r, v, swivel l)
```

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
```

Functions on perfect nested trees

```
val ? : 'a ntree → int

val ? : 'a ntree → 'a option

val ? : 'a ntree → 'a ntree
```

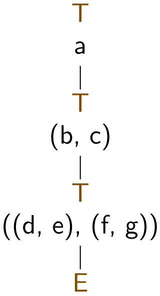
Perfect trees: depth

T
a
|
T
(b, c)
|
T
((d, e), (f, g))
|
E

1 +
1 +
1 +
0

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

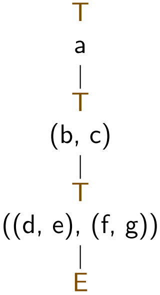
Perfect trees: top

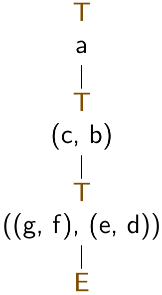


Some a

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

Perfect trees: swivel

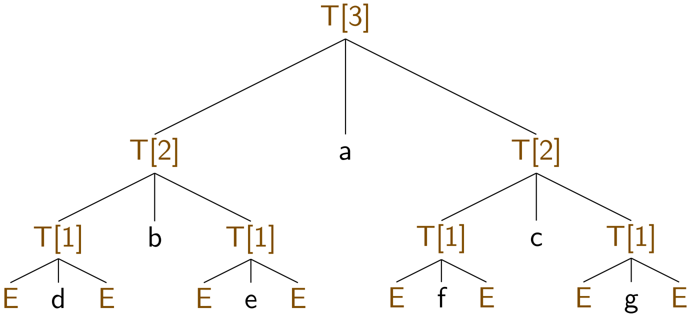





```
let rec swiv : 'a.('a→'a) → 'a ntree → 'a ntree =
  fun f t → match t with
    EmptyN → EmptyN
  | TreeN (v,t) →
    TreeN (f v,swiv (fun (x,y) → (f y, f x)) t)

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

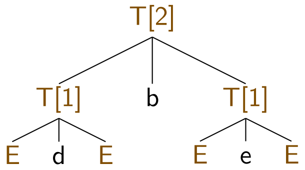
```
type z = Z
type _ s = S : 'n → 'n s
```

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

Functions on 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: $n \equiv z$

In the TreeG branch: $n \equiv m\ s$ (for some m)

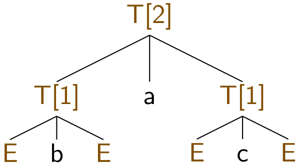
$l\ :\ (a,\ m)$ gtree

depthG $l\ :\ m$

Polymorphic recursion

The argument to the recursive call has size m (s.t. $s\ m \equiv n$)

Perfect trees (GADTs): top



a

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

Perfect trees (GADTs): depth

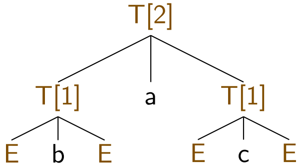
```
type ('a, _) gtree =
  EmptyG : ('a,z) gtree
| TreeG : ('a,'n) gtree * 'a * ('a,'n) gtree → ('a,'n s) gtree

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

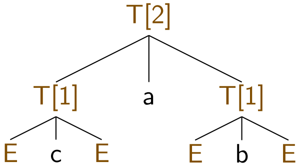
Type refinement

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)
```

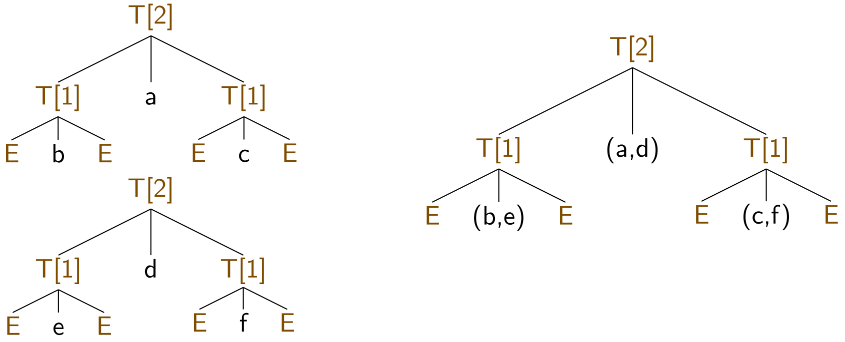
Type refinement

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 n.(a,n) gtree → (a,n) gtree → (a * a,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 n.(a,n) gtree → (a,n) gtree → (a * a,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

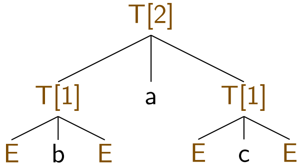
In the EmptyG branch: $n \equiv z$

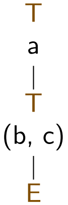
In the TreeG branch: $n \equiv m\ s$ (for some m)

EmptyG, TreeG _ produces $n \equiv z$ **and** $n \equiv m\ s$

— impossible!

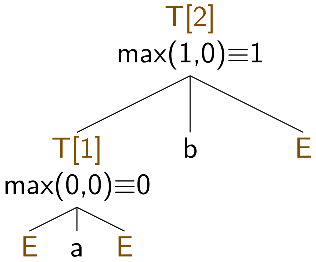
Conversions between perfect tree representations






```
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: max is one of

```
fun x _ → x
fun _ y → y
```

A typed maximum function

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

$$(\text{max}\ (a,b) \equiv c) \to a \to b \to c$$

A typed maximum function: equality

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

$a \equiv a$

A typed maximum function: a max predicate

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

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

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

A typed maximum function

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

type (_,_,_) max =
    MaxEq : ('a,'b) eql → ('a,'b,'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 Refl , _      → m
    | MaxFlip mx', _      → max mx' n m
    | MaxSuc mx' , S m'   → S (max mx' m' n)
```

A typed maximum function

```
type (_,_,_) max =
    MaxEq : ('a,'b) eql → ('a,'b,'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 Refl , _      → m
    | MaxFlip mx', _      → max mx' n m
    | MaxSuc mx' , S m'   → S (max mx' m' n)
```

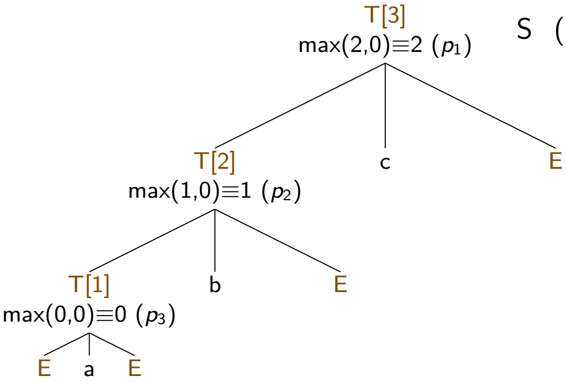
Type refinement

In the MaxEq branch:	a ≡ b, a ≡ c
	m : c
In the MaxFlip branch:	*no refinement*
In the MaxSuc branch:	a ≡ d s, c ≡ d s (for some d)
	mx' : (d, b, d) max
	m' : d
	max mx' m' n : d

Functions on depth-annotated trees

```
val ? : ('a,'n) dtree → 'n

val ? : ('a,'n s) dtree → 'a

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

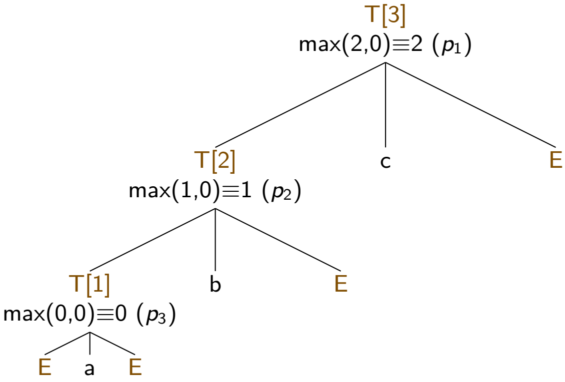
Depth-annotated trees: depth




```
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))
```

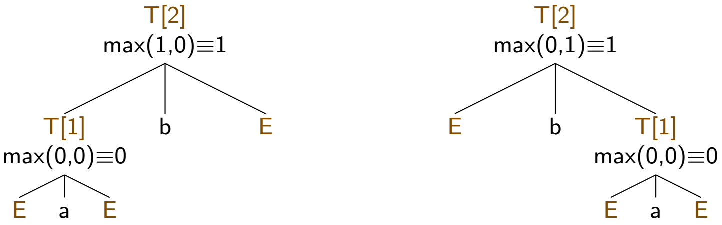
Depth-annotated trees: top



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 n.(a,n) gtree → (a,n) gtree → (a * a,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))
```

Adverts

Reflection without Remorse (Oleg Kiselyov)
3pm-4pm Monday 9th, SS03

OCaml compiler hacking
6pm-10pm Tuesday 10th, FW11

6.30pm	Generating code with polymorphic let (Oleg Kiselyov)
7pm	Pizza
7.30pm	Delve into OCaml internals