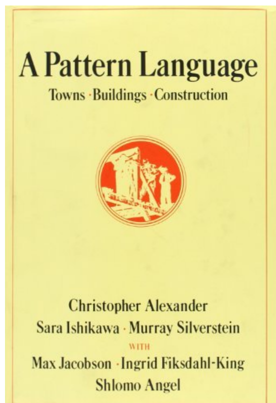
Last time: GADTs

$$a \equiv b$$

This time: some GADT programming patterns



Term inference for depth-annotated trees

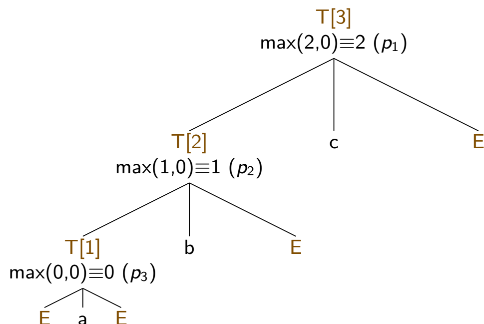
```
type ('a,_) dtree =
  EmptyD : ('a,z) dtree
| TreeD : ('a,'m) dtree
        * 'a
        * ('a,'n) dtree
        * ('m,'n,'o) max
      → ('a,'o s) dtree
```

```
  val ? : ('a,'n) dtree → 'n

  val ? : ('a,'n s) dtree → 'a

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

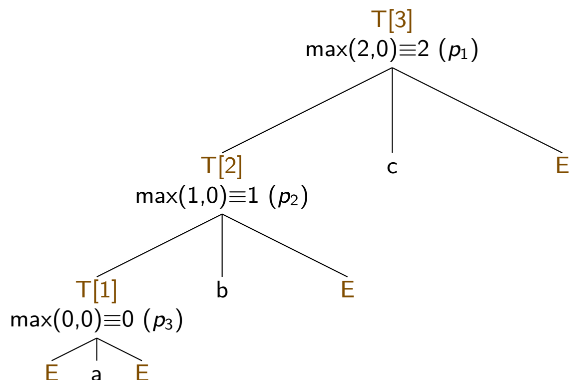
Depth-annotated trees: depth




```
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))
```

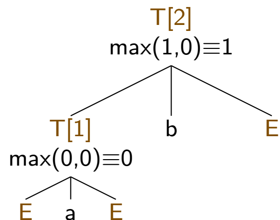
Depth-annotated trees: top

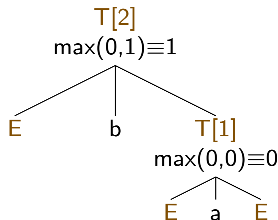


c

```
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

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

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

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

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

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 *)
```

Some GADT programming patterns

Pattern: Building GADT values

It's not always possible to determine index types statically.

For example, the depth of a tree might depend on user input.

Building GADT values: two approaches

How might `mk_dtree` build a value of the GADT type `('a,'n)dtree`?

```
let mk_dtree
  : type a n.a option → (a, n) dtree
  = ...
```

Building GADT values: two approaches

How might `mk_dtree` build a value of the GADT type `('a,'n)dtree`?

```
let mk_dtree
  : type a n.a option → (a, n) dtree
  = ✗
```

Building GADT values: two approaches

How might `mk_dtree` build a value of the GADT type `('a,'n)dtree`?

```
let mk_dtree
  : type a n.a option → (a, n) dtree
  = ✗
```

With **existentials**:

`mk_dtree` **builds** a value of type `('a,'n)dtree` for **some** `'n`.

Building GADT values: two approaches

How might `mk_dtree` build a value of the GADT type `('a,'n)dtree`?

```
let mk_dtree
  : type a n.a option → (a, n) dtree
  = ✗
```

With **existentials**:

`mk_dtree` **builds** a value of type `('a,'n)dtree` for **some** `'n`.

With **universals**:

the caller of `make_dtree` must **accept** `('a, 'n)dtree` for **any** `'n`.

Building GADT values with existentials

```
type 'a edtree = E : ('a, 'n) dtree → 'a edtree
```

```
let mk_dtree : type a n. a option → a edtree =
 function
   None → E EmptyD
 | Some v → E (TreeD (EmptyD, v, EmptyD, MaxEq Z))
```

Building GADT values with universals

```
type ('a, 'k) adtree = { k: 'n. ('a, 'n) dtree → 'k }
```

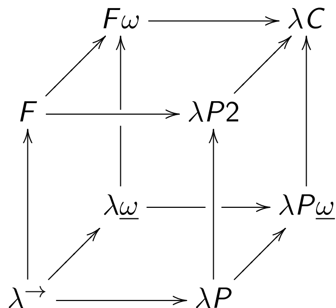
```
let mk_dtree_k: type a k.a option → (a, k) adtree → k =
 fun opt {k} → match opt with
   None → k EmptyD
 | Some v → k (TreeD (EmptyD, v, EmptyD, MaxEq Z))
```

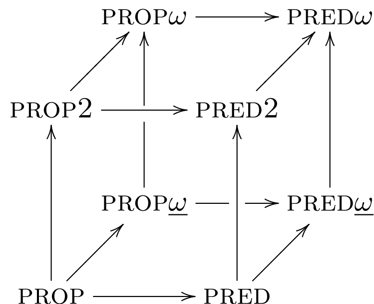
Pattern: Singleton types

Without dependent types we can't write predicates involving data.

Using one type per value allows us to simulate value indexing.

Singleton types: Lambda and logic cubes





Singleton sets bring propositional logic closer to predicate logic.

$$\forall A.B \qquad \forall\{x\}.B \qquad \forall x \in A.B$$

Singleton types

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

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

type (_,_,_) add =
    AddZ : (z,'n,'n) add
  | AddS : ('m,'n,'o) add → ('m s,'n,'o s) add
```

Pattern: Building evidence

With type refinement we learn about types by inspecting values.

Predicates should return useful *evidence* rather than `true` or `false`.

Building evidence: predicates returning bool

```
let is_empty : 'a .'a tree → bool = function
    Empty → true
  | Tree _ → false
```

```
if not (is_empty t) then
  top t
else
  None
```

Building evidence: trees

```
type _ is_zero =
  Is_zero : z is_zero
| Is_succ : _ s is_zero
```

```
let is_empty : type a n.(a,n) dtree → n is_zero =
 function
 | EmptyD → Is_zero
 | TreeD _ → Is_succ
```

```
match is_empty t with
| Is_succ → Some (topD t)
| Is_zero → None
```

Summary

Building GADT values

Singleton types

Building evidence

Next time: overloading

```
val (=) : {E:EQ} → E.t → E.t → bool
```