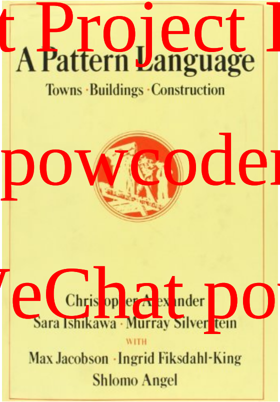
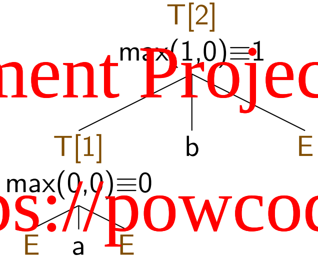
Last time: GADTs

$a \equiv b$

This time: GADT programming patterns



Recap: 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
```

Functions on depth-annotated trees

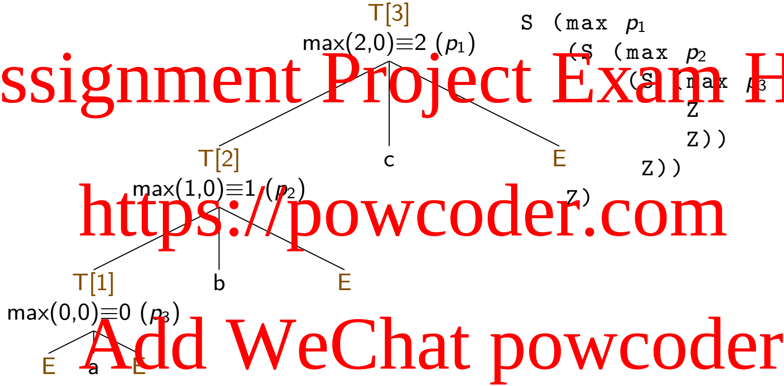

```
val ? : ('a,'n) dtree → 'n
```



```
val ? : ('a,'n s) dtree → 'a
```

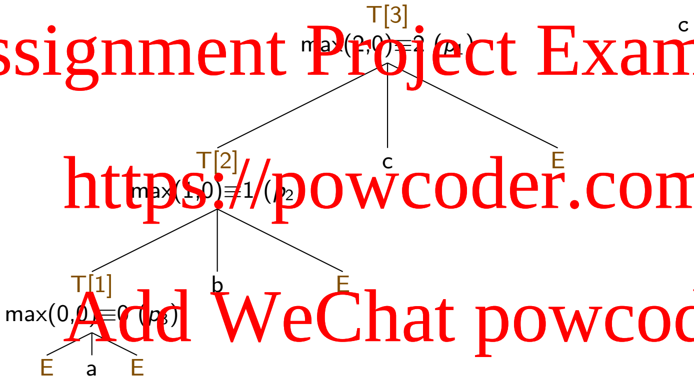
```
val ? : ('a,'n) dtree → ('a,'n) dtree
```

Depth-annotated trees: depth




```
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




```
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)
```

Recapitulation

Recapitulation: philosophical

Phantom types protect abstractions against misuse.

GADTs also protect definitions.

GADTs lead to rich types which can be viewed as propositions.

Descriptive data types lead to useful function types.

Recapitulation: technical

GADT type indexes vary across constructors.

We have families of types: a type per nat, per tree depth, etc.

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

GADTs are about type equalities (and sometimes inequalities).

Type equalities are revealed by examining data.

Compilers use the richer types to generate better code.

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

$\texttt{Eq} = \lambda\alpha{::}*.\lambda\beta{::}*.\forall\phi{::}* \Rightarrow *.\phi\ \alpha \rightarrow \phi\ \beta$

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

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

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

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 *)
```

GADT programming patterns

A new example: representing (some) JSON

JSON values

```
json       ::= true | false | string | number | null
               [] | [ json-seq ]
               {} | { json-kvseq }
json-seq   ::= json
               json , json-seq
json-kvseq ::= string : json
               string : json , json-kvseq
```

A new example: representing (some) JSON

JSON values

```
json  ::=  true | false | string | number | null
           [] | [ json-seq ]

json-seq  ::=  json
               json , json-seq
```

A JSON value

```
[ "one", true, 3.4, [[ "four" ], [null]]]
```

An "untyped" JSON representation


```
type ujson =
    UStr : string → ujson
  | UNum : float → ujson
  | UBool : bool → ujson
  | UNull : ujson
  | UArr : ujson list → ujson
```



```
[ "one", true, 3.4, [[ "four" ], [null]]]
```



```
UArr [UStr "one"; UBool true; UNum 3.4;
      UArr [UArr [UStr "four"]; UArr [UNull]]]
```

Pattern: Richly typed data

Data may have finer structure than algebraic data types can express.

GADT indexes allow us to specify constraints more precisely.

Richly typed data


```
type _ tjson =
    Str : string → string tjson
  | Num : float → float tjson
  | Bool : bool → bool tjson
  | Null : unit tjson
  | Arr : 'a tarr → 'a tjson
and _ tarr =
    Nil : unit tarr
  | :: : 'a tjson * 'b tarr → ('a*'b) tarr
```



```
Arr (Str "one" :: Bool true :: Num 3.4 ::
     Arr (Arr (Str "four" :: Nil) :: Null :: Nil) :: Nil)
```

Richly typed data


```
(* negate all the bools in a JSON value *)
let rec unegate : ujson → ujson = ...
```



```
(* negate all the bools in a JSON value *)
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
```

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 a function *make_t* build a value of a GADT type *t*?

```
let make_t : type a.string → a t = ...
```

Building GADT values: two approaches

How might a function *make_t* build a value of a GADT type *t*?

```
let make_t : type a.string → a t = ✗
```

Building GADT values: two approaches

How might a function *make_t* build a value of a GADT type *t*?

```
let make_t : type a.string → a t = ✗
```


With **existentials** `make_t` **builds** a value of type `'a t` for **some** `'a`.

Building GADT values: two approaches

How might a function *make_t* build a value of a GADT type 'a t?

```
let make_t : type a.string → a t = ✗
```


With **existentials** `make_t` **builds** a value of type `'a t` for **some** `'a`.

With **universals** the caller of `make_t` must **accept** `'a t` for **any** `'a`.

Building GADT values with existentials

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
  | UNum u → ETJson (Num u)
  | UBool b → ETJson (Bool b)
  | UNull → ETJson Null
  | UArr arr →
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with existentials

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
          (* Str s : string tjson *)
  | UNum u → ETJson (Num u)
  | UBool b → ETJson (Bool b)
  | UNull → ETJson Null
  | UArr arr →
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with existentials

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
            (* Str s : string tjson *)
  | UNum u → ETJson (Num u)     (* Num u : float tjson *)
  | UBool b → ETJson (Bool b)
  | UNull → ETJson Null
  | UArr arr →
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with existentials

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
            (* Str s : string tjson *)
  | UNum u → ETJson (Num u)     (* Num u : float tjson *)
  | UBool b → ETJson (Bool b)   (* Bool b : bool tjson *)
  | UNull → ETJson Null
  | UArr arr →
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with existentials


```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
            (* Str s : string tjson *)
  | UNum u → ETJson (Num u)       (* Num u : float tjson *)
  | UBool b → ETJson (Bool b)     (* Bool b : bool tjson *)
  | UNull → ETJson Null           (* Null : unit tjson *)
  | UArr arr →
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with existentials

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
          (* Str s : string tjson *)
  | UNum u → ETJson (Num u)             (* Num u : float tjson *)
  | UBool b → ETJson (Bool b)           (* Bool b : bool tjson *)
  | UNull → ETJson Null                 (* Null : unit tjson *)
  | UArr arr →                          (* arr' : ?a tarr *)
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with existentials

```
type etjson = ETJson : 'a tjson → etjson
type etarr = ETArr : 'a tarr → etarr

let rec tjson_of_ujson : ujson → etjson = function
    UStr s → ETJson (Str s)
          (* Str s :  string tjson *)
  | UNum u → ETJson (Num u)            (* Num u :  float tjson *)
  | UBool b → ETJson (Bool b)          (* Bool b :  bool tjson *)
  | UNull → ETJson Null                (* Null :  unit tjson *)
  | UArr arr →                         (* arr' :  ?a tarr *)
    let ETArr arr' = tarr_of_uarr arr in
    ETJson (Arr arr')                  (* Arr arr' : ?a tjson *)
and tarr_of_uarr : ujson list → etarr = function
    [] → ETArr Nil
  | j :: js →
    let ETJson j' = tjson_of_ujson j in
    let ETArr js' = tarr_of_uarr js in
    ETArr (j' :: js')
```

Building GADT values with universals

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}

let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
  | UNum u → return (Num u)
  | UBool b → return (Bool b)
  | UNull → return Null
  | UArr arr →
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Building GADT values with universals

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}

let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
           (* Str s : string tjson *)
  | UNum u → return (Num u)
  | UBool b → return (Bool b)
  | UNull → return Null
  | UArr arr →
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Building GADT values with universals

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}


let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
           (* Str s : string tjson *)
  | UNum u → return (Num u)      (* Num u : float tjson *)
  | UBool b → return (Bool b)
  | UNull → return Null
  | UArr arr →
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Building GADT values with universals

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}


let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
          (* Str s : string tjson *)
  | UNum u → return (Num u)      (* Num u : float tjson *)
  | UBool b → return (Bool b)    (* Bool b : bool tjson *)
  | UNull → return Null
  | UArr arr →
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Building GADT values with universals

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}


let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
              (* Str s : string tjson *)
  | UNum u → return (Num u)        (* Num u : float tjson *)
  | UBool b → return (Bool b)      (* Bool b : bool tjson *)
  | UNull → return Null            (* Null : unit tjson *)
  | UArr arr →
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Building GADT values with universals

```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}


let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
           (* Str s :  string tjson *)
  | UNum u → return (Num u)        (* Num u :  float tjson *)
  | UBool b → return (Bool b)      (* Bool b :  bool tjson *)
  | UNull → return Null            (* Null :  unit tjson *)
  | UArr arr →                     (* arr' :  ?a tarr *)
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

Building GADT values with universals

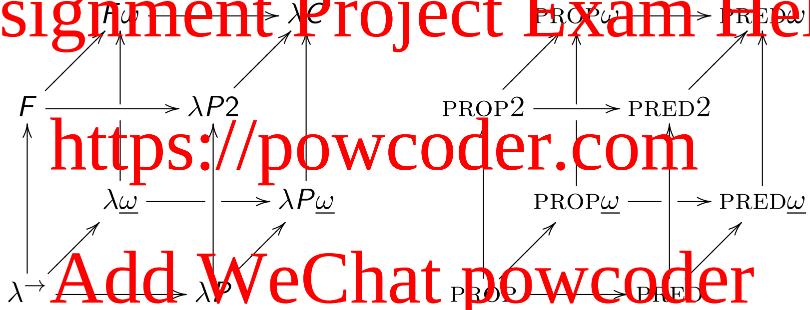
```
type 'k atjson = {k: 'a. 'a tjson → 'k}
type 'k atarr = {k: 'a. 'a tarr → 'k}

let rec tjson_of_ujson : ujson → 'k atjson → 'k =
  fun j {k=return} → match j with
    UStr s → return (Str s)
           (* Str s : string tjson *)
  | UNum u → return (Num u)       (* Num u : float tjson *)
  | UBool b → return (Bool b)     (* Bool b : bool tjson *)
  | UNull → return Null           (* Null : unit tjson *)
  | UArr arr →                    (* arr' : ?a tarr *)
    tarr_of_uarr arr {k = fun arr' →
    return (Arr arr') }           (* Arr arr' : ?a tjson *)
and tarr_of_uarr : ujson list → 'k atarr → 'k =
  fun jl {k=return} → match jl with
    [] → return Nil
  | j :: js →
    tjson_of_ujson j {k = fun j' →
    tarr_of_uarr js {k = fun js' →
    return (j' :: js') }}
```

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,'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
```



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

Pattern: Separating types and data

Entangling proofs and data can lead to redundant, inefficient code.

Separate proofs make data reusable and help avoid slow traversals.

Separating types and data

```
type _ tyjson =
    TyStr : string tyjson
  | TyNum : float tyjson
  | TyBool : bool tyjson
  | TyNull : 'a tyjson → 'a option tyjson
  | TyArr : 'a tyarr → 'a tyjson
and _ tyarr =
    TyNil : unit tyarr
  | :: : 'a tyjson * 'b tyarr → ('a * 'b) tyarr
```


Entangled

```
Arr (Str "one" :: Bool true :: Num 3.4 ::
    Arr (Arr (Str "four" :: Nil) :: Null :: Nil) :: Nil)
```


Disentangled

```
TyArr (TyStr :: TyBool :: TyNum ::
          TyArr (TyArr (TyStr :: TyNil)
                 :: TyNull TyBool :: TyNil) :: TyNil)
("one", (true, (3.4, (((("four", ()), (None, ())), ())
   ))))
```

Separating types and data

The `negate` function, entangled


```
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
```

Separating types and data

The `negate` function, disentangled


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

Separating types and data

The `negate` **function, disentangled and "staged"**


```
let rec negateDS : type a.a tyjson → a → a =
   function
    TyBool → (function false → true
                     | true → false)
  | TyArr a → negate_arrDS a
  | _ → (fun v → v)
and negate_arrDS : type a.a tyarr → a → a = function
    TyNil → (fun () → ())
  | j :: js → let n = negateDS j
              and ns = negate_arrDS js in
                (fun (a, b) → (n a, ns b))
```

Separating types and data: verifying data

```
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
  | TyNull _, UNull → Some None
  | TyNull j, v → (match unpack_ujson j v with
                        Some v → Some (Some v)
                      | None → None)
  | TyArr a, UArr arr → unpack_uarr a arr
  | _ → None
and unpack_uarr :
  type a.a tyarr → ujson list → a option =
  fun ty v → match ty, v with
    TyNil, [] → Some ()
  | j :: js, v :: vs →
      (match unpack_ujson j v, unpack_uarr js vs with
          Some v', Some vs' → Some (v', vs')
        | _ → None)
  | _ → None
```

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
```

Representing types built from strings and arrays

```
type _ str_tyjs =
    SStr : string str_tyjs
  | SArr : 'a str_tyarr → 'a str_tyjs
and 'a str_tyarr =
    SNil : unit str_tyarr
  | :: : 'a str_tyjs * 'b str_tyarr → ('a*'b) str_tyarr
```

Building evidence: JSON

Determining whether a type is built from strings and arrays

```
let rec is_stringy : type a.a tyjson → a str_tyjs option
  =
  function
    TyStr → Some SStr
  | TyNum → None
  | TyBool → None
  | TyNull _ → None
  | TyArr arr → match is_stringy_array arr with
                    None → None
                  | Some sarr → Some (SArr sarr)
and is_stringy_array :
 type a.a tyarr → a str_tyarr option =
 function
    TyNil → Some SNil
  | x :: xs →
    match is_stringy x, is_stringy_array xs with
        Some x, Some xs → Some (x :: xs)
      | _ → None
```

Building evidence: JSON (entangled)

Determining whether a value is built from strings and arrays

```
let rec is_stringyV : type a.a tjson → a str_tyjs option
  =
 function
    Str _ → Some SStr
  | Num _ → None
  | Bool _ → None
  | Null → None
  | Arr arr → match is_stringy_arrayV arr with
                None → None
              | Some sarr → Some (SArr sarr)
and is_stringy_arrayV :
 type a.a tarr → a str_tyarr option =
 function
    Nil → Some SNil
  | x :: xs →
    match is_stringyV x, is_stringy_arrayV xs with
        Some x, Some xs → Some (x :: xs)
      | _ → None
```

Summary

Richly typed data

Building GADT values

Singleton types

Separating types and data

Building evidence

Next time: monads etc.

$\gg=$