Last time: monads (etc.)

This time: generic programming

$$\texttt{val}\ \text{show} : \text{'a} \rightarrow \text{string}$$

Generic functions

Data

unit

```
()
```

booleans

```
false, true
```

ints

```
...,-2,-1,0,1,2,...
```

floats

```
0.0, nan, 2.2e-308,
3.1416, infinity
```

sums

```
L (), R 3
```

pairs

```
(false, 3.0)
```

records

```
{x = {y = ()}; z = true}
```

lists

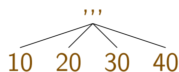
```
[2; 4; 6; 8],
[false; true], [[]; []]
```

Data as trees

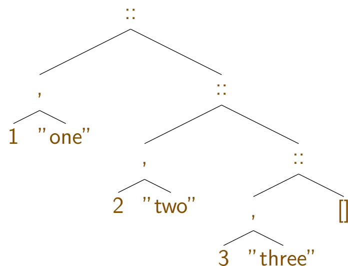
L

()

```
L ()
```




```
(10,20,30,40)
```




```
[(1, "one"); (2, "two"); (3, "three")]
```

Operations defined on (most) data

equality
$'a \to 'a \to \text{bool}$

hashing
$'a \to \text{int}$

ordering
$'a \to 'a \to \text{int}$

mapping
$('b \to 'b) \to 'a \to 'a$

querying
$('b \to \text{bool}) \to 'a \to 'b\ \text{list}$

pretty-printing
$'a \to \text{string}$

parsing
$\text{string} \to 'a$

serialising
$'a \to \text{string}$

sizing
$'a \to \text{int}$

Parameterising by shape

Generic functions and parametricity

Some built-in OCaml functions are incompatible with parametricity:

```
val (=) : 'a → 'a → bool

val hash : 'a → int

val from_string : string → int → 'a
```

Generic functions and parametricity

Some built-in OCaml functions are incompatible with parametricity:

```
val (=) : 'a → 'a → bool
```

```
val hash : 'a → int
```

```
val from_string : string → int → 'a
```

How might we do better? Parameterise by **shape**:

```
val (=) : 'a data → 'a → 'a → bool
```

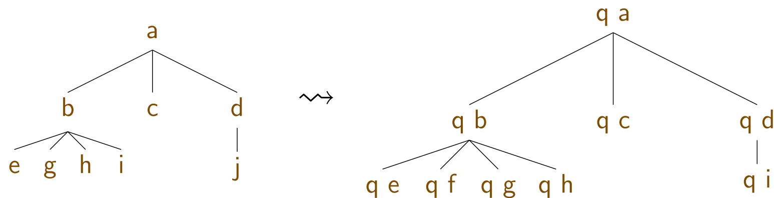
```
val hash : 'a data → 'a → int
```

```
val from_string : 'a data → string → int → 'a
```

Traversing trees

Plan: use shapes to describe how to traverse trees.

As we traverse, we'll apply a function at every node.

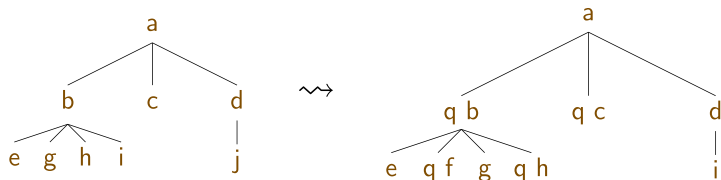


We'll need a way to traverse data

Traversing trees

Plan: use shapes to describe how to traverse trees.

As we traverse, we'll apply a function at every applicable node.



We'll need a way to traverse data

We'll need a way to determine the type of data

Type-indexed values

Idea: represent OCaml types by values of some indexed type t:

```
val int : int t
val bool : bool t
val ( * ) : 'a t → 'b t → ('a * 'b) t
val list : 'a t → 'a list t
val option : 'a t → 'a option t
(* etc. *)
```

Type	**Representation**	**Representation type**
int	int	int t
bool	bool	bool t
int * bool	int * bool	int * bool t
int option list	list (option int)	int option list t

Testing type equality

Type indexed values for type equality

```
type 'a typeable

module Typeable :
sig
  val int : int typeable
  val bool : bool typeable
  val list : 'a typeable → 'a list typeable
  val ( * ) : 'a typeable → 'b typeable →
     ('a * 'b) typeable
  (* ... *)
end
```

```
# Typeable.(list (int * bool));;
- : (int * bool) list typeable = ...
```

Determining type equality at runtime

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

val (=~=) : 'a typeable → 'b typeable → ('a,'b) eql option
```

Representing types

```
type _ typeable =
  Int : int typeable
| Bool : bool typeable
| List : 'a typeable → 'a list typeable
| Option : 'a typeable → 'a option typeable
| Pair : 'a typeable * 'b typeable → ('a * 'b) typeable
```

Implementing type equality

```
let rec eqty :
 type a b.a typeable → b typeable → (a,b) eql option =
 fun l r → match l, r with
    Int, Int → Some Refl
  | Bool, Bool → Some Refl
  | List s, List t →
     (match eqty s t with
        Some Refl → Some Refl
      | None → None)
  | Option s, Option t →
     (match eqty s t with
        Some Refl → Some Refl
      | None → None)
  | Pair (s1, s2), Pair (t1, t2) →
     (match eqty s1 t1, eqty s2 t2 with
        Some Refl, Some Refl → Some Refl
      | _ → None)
  | _ → None
```

Implementing type equality

```
let rec eqty :
 type a b.a typeable → b typeable → (a,b) eql option =
 fun l r → match l, r with
     Int, Int → Some Refl
   | Bool, Bool → Some Refl
   | List s, List t →
      (match eqty s t with
         Some Refl → Some Refl
       | None → None)
   | Option s, Option t →
      (match eqty s t with
         Some Refl → Some Refl
       | None → None)
   | Pair (s1, s2), Pair (t1, t2) →
      (match eqty s1 t1, eqty s2 t2 with
         Some Refl, Some Refl → Some Refl
       | _ → None)
   | _ → None
```

Problem: this representation has no support for user-defined types.

Extensible variants

Defining

```
type 'a t = ..
```

Extending

```
type 'a t +=
  A : int list → int t
| B : float list → 'a t
```

Constructing

```
A [1;2;3]  (* No different to standard variants *)
```

Matching

```
let f : type a. a t → string = function
  A _ → "A"
| B _ → "B"
| _ → "unknown" (* All matches must be open *)
```

Representing types, extensibly

```
type _ type_rep = ..

type 'a typeable = {
  type_rep : 'a type_rep;
  eqty : 'b. 'b type_rep → ('a, 'b) eql option;
}

type _ type_rep += Int : int type_rep

let eq_int :
 type b. b type_rep → (int, b) eql option =
 function Int → Some Refl | _ → None

let int = { type_rep = Int; eqty = eq_int }
```

Representing types, extensibly

```
type _ type_rep = ..

type 'a typeable = {
  type_rep : 'a type_rep;
  eqty : 'b. 'b type_rep → ('a, 'b) eq option;
}

type _ type_rep += List : 'a type_rep → 'a list type_rep

let eq_list :
 type a b.a typeable → b type_rep → (a list ,b) eq option =
 fun t → function
     List a → (match t.eqty a with
                 Some Refl → Some Refl
               | None → None)
   | _ → None

let list a = { type_rep = List a.type_rep;
               eqty = fun t → eq_list a t }
```

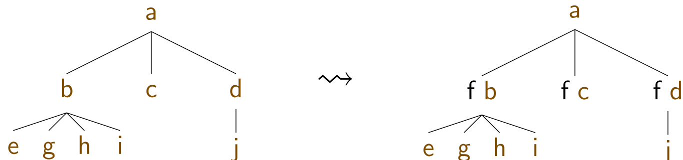
Implementing type equality, extensibly

```
let (=~=) :
 type a b.a typeable → b typeable → (a,b) eql option =
  fun a b → a.eqty b.type_rep
```

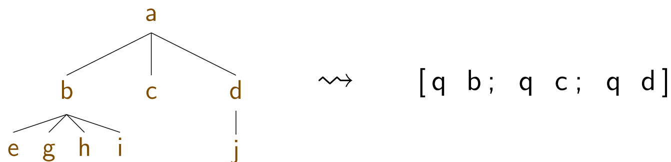
Traversing datatypes

Traversing datatypes

gmapT



gmapQ

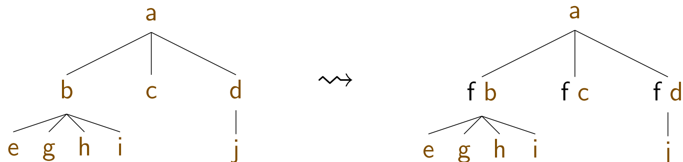


Type indexed values for traversable types

```
type 'a data

module Data :
sig
  val int : int data
  val bool : bool data
  val list : 'a data → 'a list data
  val ( * ) : 'a data → 'b data →
      ('a * 'b) data
  (* ... *)
end
```

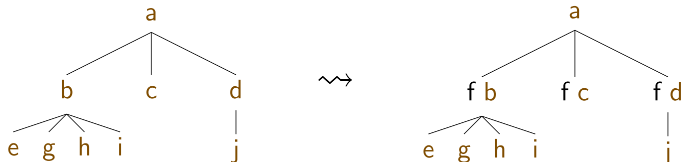
```
# Data.(list (int * bool));;
- : (int * bool) list data = ...
```

Polymorphic types for generic traversals: gmapT



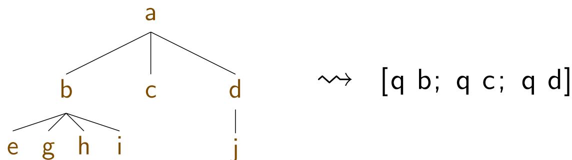

```
val gmapT :
  'a data → (∀'b.'b data → 'b → 'b) → 'a → 'a
```

Polymorphic types for generic traversals: gmapT



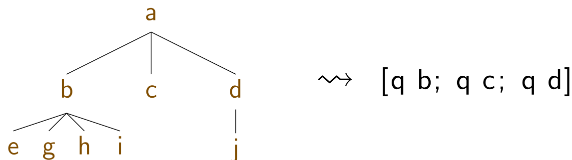

```
type gmapT_arg = { f : 'b. 'b data → 'b → 'b; }

val gmapT :
  'a data → gmapT_arg → 'a → 'a
```

Polymorphic types for generic queries: gmapQ




```
val gmapQ :
  'a data → (∀'b.'b data → 'b → 'u) → 'a → 'u list
```

Polymorphic types for generic queries: gmapQ




```
type 'u gmapQ_arg = { q : 'b. 'b term → 'b → 'u }

val gmapQ :
  'a data → 'u gmapQ_arg → 'a → 'u list
```

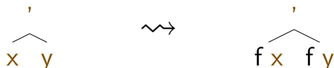
Traversing datatypes: primitive types

$$x \quad \rightsquigarrow \quad x$$

gmapT int f

```
let gmapT_int f x = x

let gmapQ_int q x = []
```

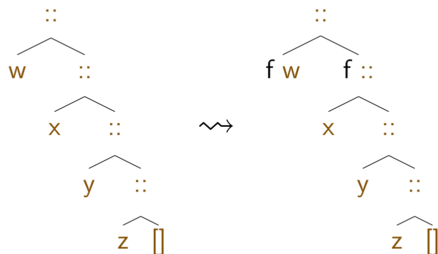
Traversing datatypes: pairs



gmapT (a ∗ b) f

```
let gmapT_pair { f } (x, y) = (f a x, f b y)

let gmapQ_pair { q } (x, y) = [q a x; q b y]
```

Traversing datatypes: lists



let l = list a in gmapT l f

```
let gmapT_list {f} = function
    [] → []
  | x :: xs → f a x :: f l xs

let gmapQ_list {q} = function
    [] → []
  | x :: xs → [q a x; q l xs]
```

Type indexed values for traversals

```
type 'a data = {
  typeable : 'a typeable;
  gmapT : gmapT_arg → 'a → 'a;
  gmapQ : 'u. 'u gmapQ_arg → 'a → 'u list;
}
and gmapT_arg = { f : 'b. 'b data → 'b → 'b; }
and 'u gmapQ_arg = { q : 'b. 'b data → 'b → 'u }
```

Type indexed values for traversals

```
type 'a data = {
  typeable : 'a typeable;
  gmapT : gmapT_arg → 'a → 'a;
  gmapQ : 'u. 'u gmapQ_arg → 'a → 'u list;
}
and gmapT_arg = { f : 'b. 'b data → 'b → 'b; }
and 'u gmapQ_arg = { q : 'b. 'b data → 'b → 'u }

module Data = struct
  let int = {
  { typeable = Typeable.int;
    gmapT = gmapT_int;
    gmapQ = gmapQ_int; }

  let ( * ) a b =
  { typeable = Typeable.pair a.typeable b.typeable;
    gmapT = gmapT_pair;
    gmapQ = gmapQ_pair; }

  (* etc. *)
```

Generic schemes

Generic maps, bottom up

```
let rec everywhere : 'a.'a data → gmapT_arg → 'a → 'a =
  fun t f x → f.f t
      (t.gmapT { f = fun t → everywhere t f } x)
```

Generic maps, top down

```
let rec everywhere' : 'a.'a data → gmapT_arg → 'a → 'a =
  fun t f x →
    (t.gmapT { f = fun t y → everywhere' t f (f.f t y) } x)
```

Generic maps with a stop condition

```
let rec everywhereBut : 'a.
'a data → bool gmapQ_arg → gmapT_arg → 'a → 'a =
 fun t q f x →
  if q.q t x then x
  else f.f t
         (t.gmapT {f = fun t → everywhereBut t q f} x)
```

Using generic maps

```
val everywhere : 'a data → gmapT_arg → 'a → 'a
```

```
let mkT : type b. b typeable → (b → b) → gmapT_arg =
  fun b g →
    let f : type a. a data → a → a =
      fun a x → match a.typeable =~= b with
      | Some Refl → g x
      | _         → x
    in { f }
```

```
everywhere
  (list (bool * int))
  (mkT Typeable.int succ)
  [(false, 1); (false, 2); (true, 3)]
```

Generic queries

```
let rec everything : 'a 'r.
  'a data → ('r → 'r → 'r) → 'r gmapQ_arg → 'a → 'r =
  fun t k q x →
    List.fold_left k (q.q t x)
    (t.gmapQ {q = fun t → everything t k q} x)
```

```
let rec everythingBut : 'a 'r.
  'a data → ('r → 'r → 'r) → ('r * bool) gmapQ_arg → 'a → 'r =
  fun t k q x →
   let (v, stop) = q.q t x in
   if stop then v
   else List.fold_left k v
        (t.gmapQ {q = fun t → everythingBut t k q} x)
```

Using generic queries

```
val everything :
    'a data → ('r → 'r → 'r) → 'r gmapQ_arg → 'a → 'r

let mkQ :
 type b r.b typeable → r → (b → r) → r gmapQ_arg =
 fun t r br →
  let q : type a. a data → a → r =
   fun a x →
    match a.typeable =~= t with
     Some Refl → br x
    | None → r
  in { q }

everything (list (int * bool))
  (@) (mkQ Typeable.bool [] (fun x → [x]))
   [(1, false); (2, true)]
```

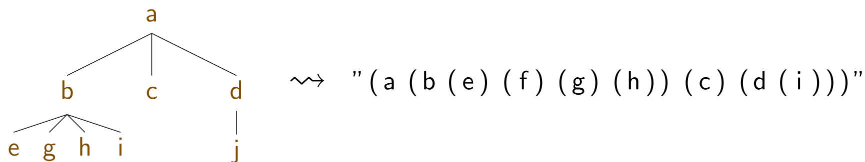
Generic printing

Representing constructors

Add an additional field to data for distinguishing constructors:

```
type constr = string

type 'a data = {
  typeable : 'a typeable;
  gmapT : gmapT_arg → 'a → 'a;
  gmapQ : 'u. 'u gmapQ_arg → 'a → 'u list;
  constructor: 'a → constr;
}
```

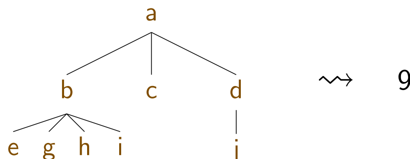
A generic printing function




```
let rec gshow : 'a. 'a data → 'a → string =
  fun t v →
    "("^ t.constructor v
    ^ String.concat " " (t.gmapQ {q = gshow} v)
    ^ ")"
```

Generic sizing

Computing value size generically




```
let rec gsize : 'a. 'a data → 'a → int =
  fun t v → 1 + sum (t.gmapQ {q = gsize} v)
```

```
gsize int 3

gsize (list int) [1;2;3]

gsize (list (int * bool))
   [(1,false); (2,false); (3,true)]
```

Remaining problems

Passing shapes around is **awkward**

```
everywhere
  (list (bool * int))
  (mkT Typeable.int succ)
  [(false, 1); (false, 2); (true, 3)]
```

Generic traversals are **slow**

```
let gmapT_pair { f } (x, y) = (f a x, f b y)
```

Remaining problems

Passing shapes around is awkward. Solution: **implicits**

```
everywhere
   (mkT                     succ)
   [(false, 1); (false, 2); (true, 3)]
```

Generic traversals are **slow**.

```
let gmapT_pair { f } (x, y) = (f a x, f b y)
```

Remaining problems

Passing shapes around is awkward. Solution: **implicits**

```
everywhere

  (mkT                    succ)
  [(false, 1); (false, 2); (true, 3)]
```

Generic traversals are slow. Solution: **staging**.

```
let gmapT_pair_int_bool f (x, y) = (f x, y)
```

Next time: staging

`.< e >.`