Last time: monads (etc.)

Recap

monads ($\gg=$)

```
let x₁ = e₁ in
let x₂ = e₂ in
   ...
let xₙ = eₙ in
  e
```

arrows ($\ggg$, `first`)

```
let x₁ = C₁ e₁ in
let x₂ = C₂ e₂ in
   ...
let xₙ = Cₙ eₙ in
  e
```

applicatives ($\otimes$)

```
let x₁ = e₁
and x₂ = e₂
   ...
and xₙ = eₙ in
  e
```

monoids ($+\!\!+$)

```
e₁ ;
e₂ ;
... ;
eₙ
```

parameterised monads and applicatives

$\{P\}\ C\ \{Q\}$

Arrows and monads: not every arrow is a monad

```
module Phantom_monoid_arrow (M: MONOID)
  : ARROW with type ('a, 'b) t = M.t =
struct
  type ('a, 'b) t = M.t
  let arr _ = M.zero
  let (>>>) f g = M.(f ++ g)
  let first f = f
end
```

Arrows and applicatives again

Reversing effect order with applicatives

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end
```

Reversing effect order with monads . . .

Reversing effect order with arrows. . .

Arrows and applicatives again

Reversing effect order with applicatives

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end
```

Reversing effect order with monads . . .

. . . is impossible, because computations have *control dependencies*

Reversing effect order with arrows. . .

Arrows and applicatives again

Reversing effect order with applicatives

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end
```

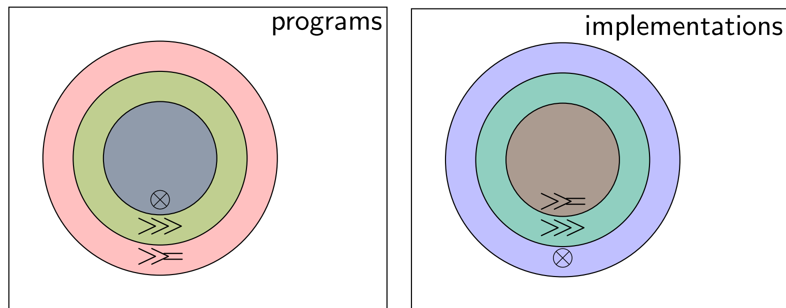
Reversing effect order with monads ...

... is impossible, because computations have *control dependencies*

Reversing effect order with arrows...

... is impossible, because computations have *data dependencies*

Applicatives vs arrows vs monads



Some monadic programs can't be written with arrows

e.g. `uncurryM`

Some arrow programs can't be written with applicatives

e.g. `fresh_name`

Some applicative instances can't be written as arrows

e.g. `Dual_applicative`

Some arrow instances can't be written as monads

e.g. `Phantom_monoid_arrow`

This time: generic programming

```
val show : 'a → string
```

Generic functions

Data types

unit

```
type unit =
 Unit : unit
```

booleans

```
type bool =
   False: bool
 | True : bool
```

natural numbers

```
type nat =
    Zero: nat
  | Succ: nat -> nat
```

sums

```
type ('a,'b) sum =
  Left : 'a -> ('a,'b) sum
| Right: 'b -> ('a,'b) sum
```

pairs

```
type ('a,'b) pair =
 Pair: 'a * 'b -> ('a,'b) pair
```

lists

```
type 'a list =
  Nil : 'a list
| Cons: 'a * 'a list -> 'a list
```

Data type operations: formatting

unit

```
let string_of_unit : unit -> string = function
  Unit -> "Unit"
```

booleans

```
let string_of_bool : bool -> string = function
   False -> "False"
 | True  -> "True"
```

natural numbers

```
let rec string_of_nat : nat -> string = function
   Zero   -> "Zero"
 | Succ n -> "(Succ "^ string_of_nat n ^")"
```

Data type operations: formatting (continued)

sums

```
let string_of_sum :
 ('a -> string) -> ('b -> string) -> ('a,'b) sum -> string =
 fun l r -> function
   Left x -> "(Left "^ l x ^")"
 | Right y -> "(Right "^ r y ^")"
```

pairs

```
let string_of_pair :
 ('a -> string) -> ('b -> string) -> ('a,'b) pair -> string =
 fun l r -> function
   Pair (x, y) -> "(Pair "^ l x ^","^ r y ^")"
```

lists

```
let rec string_of_list :
 ('a -> string) -> 'a list -> string =
 fun a -> function
   Nil -> "Nil"
 | Cons (x,xs) -> "(Cons "^ a x ^","^ string_of_list a y ^")"
```

Operations defined on (most) data

equality

'a → 'a → bool

hashing

'a → int

ordering

'a → 'a → int

mapping

('b → 'b) → 'a → 'a

querying

('b → bool) → 'a → 'b list

pretty-printing

'a → string

parsing

string → 'a

serialising

'a → string

sizing

'a → int

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? Pass a description of the data **shape**:

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

val hash : 'a data → 'a → int

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

Data shape descriptions: 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. *)
```

`int` is represented by a value

```
int : int t
```

`int * bool` is represented by a value

```
int * bool : int * bool t
```

`int option list` is represented by a value

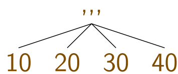
```
list (option int): int option list t
```

(etc.)

Data as trees

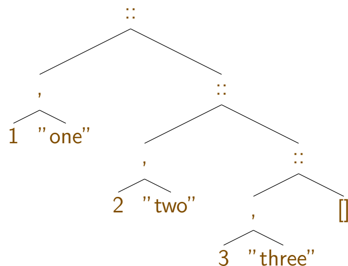
L

()




```
L ()
```

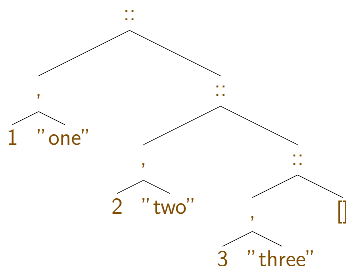
```
(10,20,30,40)
```




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

Generic operations: three questions about data

1. **What type is this data?**
2. **What are its subnodes?**
3. **What about the recursive case?**



1. Examining types

A data description for type equality

```
type 'a typeable
```

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

```
# int =~= float
- : (int, float) eql option = None

# int =~= int
- : (int, int) eql option = Some Refl
```

Type indexed values for type equality

```
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 = ...
```

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

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

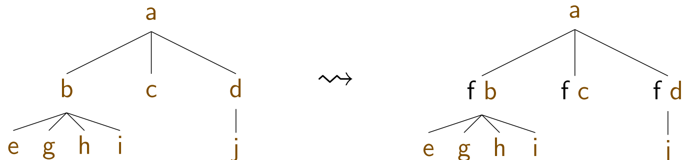
```
val (=~=) : 'a typeable → 'b typeable → ('a,'b) eql option

let (=~=) a b = a.eqty b.type_rep
```

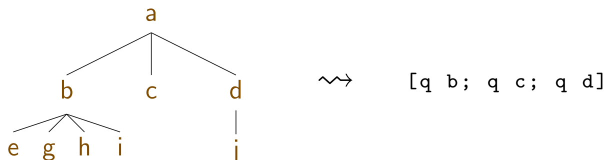
2. Accessing subnodes

Traversing datatypes

gmapT



gmapQ



A data description for accessing subnodes

```
type 'a data
```

```
val gmapT :
  ('s. 's data -> 's -> 's) -> ('t. 't data -> 't -> 't)
```

```
val gmapQ :
 ('s. 's data -> 's -> 'u) -> ('t. 't data -> 't -> 'u list)
```

A data description for accessing subnodes

```
type 'a data
```

```
type genericT = { t: 't. 't data -> 't -> 't }
val gmapT : genericT -> genericT
```

```
type 'u genericQ = { q: 't. 't data -> 't -> 'u }
val gmapQ : 'u genericQ -> 'u list genericQ
```

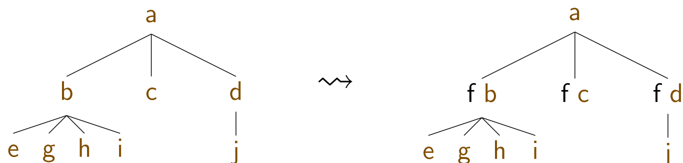
Type indexed values for accessing subnodes

```
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 = ...
```

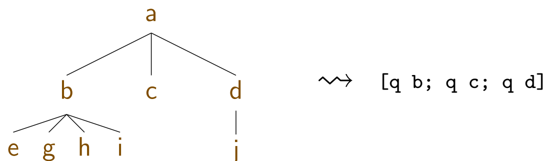
Polymorphic types for generic traversals: gmapT




```
type genericT = { t: 't. 't data -> 't -> 't }

val gmapT : genericT -> genericT
```

Polymorphic types for generic queries: gmapQ




```
type 'u genericQ = { q: 't. 't data -> 't -> 'u }

val gmapQ : 'u genericQ -> 'u list genericQ
```

Traversing datatypes: primitive types

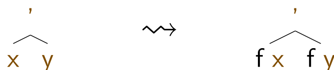
$\texttt{x} \rightsquigarrow \texttt{x}$

gmapT int f

```
let gmapT_int f x = x

let gmapQ_int q x = []
```

Traversing datatypes: pairs



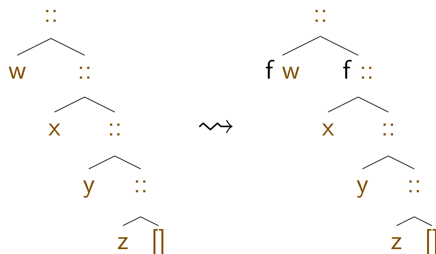

```
gmapT (a * b) f
```

Given `data` values `a` and `b` describing the type parameters:

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

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

Traversing datatypes: lists



Given a `data` value `a` describing the type parameter:

```
let gmapT_list {t=f} = function
    [] → []
  | x :: xs → f a x :: f (list a) xs

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

Type indexed values for traversals

```
type 'a data = {
  typeable : 'a typeable;
  gmapT : genericT -> 't -> 't;
  gmapQ : 'u. 'u genericQ -> 't -> 'u list;
}
```

Type indexed values for traversals

```
type 'a data = {
  typeable : 'a typeable;
  gmapT : genericT -> 't -> 't;
  gmapQ : 'u. 'u genericQ -> 't -> 'u list;
}


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. *)
```

3. Handling recursion

Generic maps, bottom up

```
let rec everywhere : genericT -> genericT =
  fun f ->
   { t =
       fun data x ->
        f.t data
          ((gmapT (everywhere f)).t data x) }


(* everywhere f x = f ((gmapT (everywhere f)) x) *)
```

Generic maps, top down

```
let rec everywhere' : genericT -> genericT =
  fun f ->
   { t =
       fun data x ->
        (gmapT (everywhere' f)).t data
           (f.t data x) }

(* everywhere' f x = gmapT (everywhere' f) (f x) *)
```

Generic maps with a stop condition

```
let rec everywhereBut :
 bool genericQ -> genericT -> genericT =
 fun stop f ->
  { t =
      fun data x ->
       if stop.q data x then x
       else
        f.t data
          ((gmapT (everywhereBut stop f)).t data x) }
```

```
(* everywhereBut stop f x =
      if stop x then x
      else f ((gmapT (everywhereBut stop f)) x) *)
```

Using generic maps

```
val everywhere : genericT -> genericT
```

```
let mkT : type t. t typeable -> (t -> t) -> genericT =
  fun t g ->
    { t = fun data x ->
          match data.typeable.eqty t.type_rep with
          | Some Refl -> g x
          | _         -> x }
```

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

Generic queries

```
let rec everything :
  'r. ('r -> 'r -> 'r) -> 'r genericQ -> 'r genericQ =
  fun h g ->
    { q =
        fun data x ->
          let f = g.q data x in
          List.fold_left h f
            ((gmapQ (everything h g)).q data x) }
```

```
let rec everythingBut :
  'r. ('r -> 'r -> 'r) -> ('r * bool) genericQ -> 'r genericQ =
  fun h stop ->
   { q =
      fun data x ->
       match stop.q data x with
       | v, true -> v
       | v, false ->
         List.fold_left h v
          ((gmapQ (everythingBut h stop)).q data x) }
```

Using generic queries

```
val everything :
  ('r -> 'r -> 'r) -> 'r genericQ -> 'r genericQ =
```

```
let mkQ :
  type t u. t typeable -> u -> (t -> u) -> u genericQ =
  fun t u g ->
    { q = fun data x ->
          match data.typeable.eqty t.type_rep with
          | Some Refl -> g x
          | _         -> u }
```

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

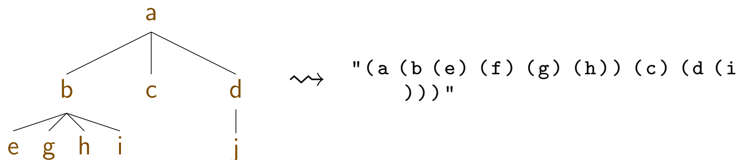
Generic printing

Representing constructors

Add an additional field to `data` for distinguishing constructors:

```
type 'a data = {
  typeable : 'a typeable;
  gmapT : genericT -> 't -> 't;
  gmapQ : 'u. 'u genericQ -> 't -> 'u list;
  constructor: 'a → string;
}
```

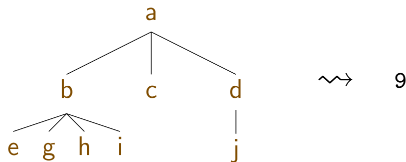
A generic printing function




```
let rec gshow : string genericQ =
 { q =
    fun data v ->
      "("^ data.constructor v
      ^ String.concat " " ((gmapQ gshow).q data v)
      ^ ")" }
```

Generic sizing

Computing value size generically




```
let rec gsize : int genericQ =
 { q = fun data v -> 1 + sum ((gmapQ gsize).q data v) }
```

```
(* gsize v -> 1 + sum ((gmapQ 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 { t=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 { t=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 >.
```