

Last time: monads (etc.)

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Recap

monads ($\gg=$)

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

arrows ($\gg>$, 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
```

<https://powcoder.com>

Add WeChat powcoder

parameterised monads and applicatives

$\{P\} \vdash \{Q\}$

Arrows and monads: not every arrow is a monad

Assignment Project Exam Help

```
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
```

<https://powcoder.com>

Add WeChat powcoder

Arrows and applicatives again

Reversing effect order with applicatives

```
module DualApplicative (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...

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Arrows and applicatives again

Reversing effect order with applicatives

```
module DualApplicative (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...

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Arrows and applicatives again

Reversing effect order with applicatives

```
module DualApplicative (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...

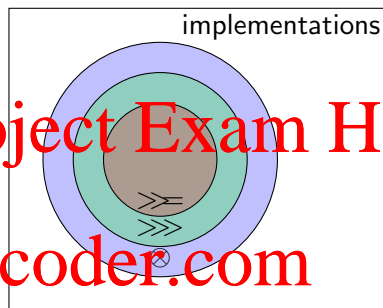
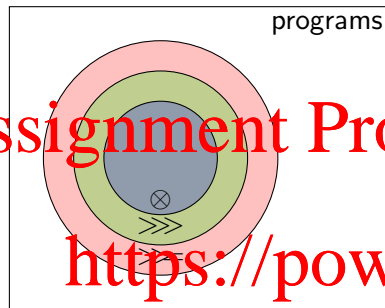
... is impossible, because computations have *data dependencies*

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Applicatives vs arrows vs monads



Assignment Project Exam Help

<https://powcoder.com>

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

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

Generic functions

Add WeChat powcoder

Data types

unit

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

sums

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

booleans

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

pairs

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

natural numbers

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

lists

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

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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 ^ ")"
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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`

pretty-printing

`'a → string`

hashing

`'a → int`

parsing

`string → 'a`

ordering

`'a → 'a → int`

serialising

`'a → string`

mapping

`('b → 'b) → 'a → 'a`

sizing

`'a → int`

querying

`('b → bool) → 'a → 'b list`

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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
```

<https://powcoder.com>

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
```

Add WeChat powcoder

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.)

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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

Generic operations: three questions about data

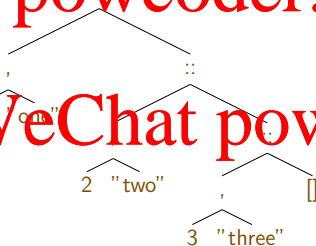
1. What type is this data?

2. What are its subnodes?

3. What about the recursive case?

<https://powcoder.com>

Add WeChat powcoder



Assignment Project Exam Help

<https://powcoder.com>

1. Examining types

Add WeChat powcoder

A data description for type equality

Assignment Project Exam Help

```
type 'a typeable
```

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

<https://powcoder.com>

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

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

Add WeChat powcoder

Type indexed values for type equality

Assignment Project Exam Help

```
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
```

<https://powcoder.com>

Add WeChat powcoder

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

Assignment Project Exam Help

```
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
```

<https://powcoder.com>

Add WeChat powcoder

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
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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 *)
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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 }
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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 fl → Some fl  
             | None → None)
```

```
  | _ → None
```

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

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

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

<https://powcoder.com>

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

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

Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

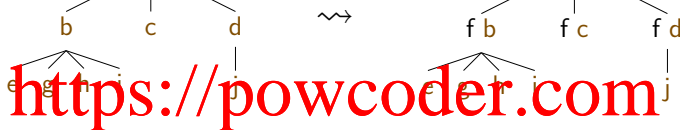
2. Accessing subnodes

Add WeChat powcoder

Traversing datatypes

gmapT

Assignment Project Exam Help



gmapQ



A data description for accessing subnodes

Assignment Project Exam Help

```
type 's data
```

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

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

Add WeChat powcoder

A data description for accessing subnodes

Assignment Project Exam Help

```
type 'a data
```

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

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

<https://powcoder.com>
Add WeChat powcoder

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));;
```

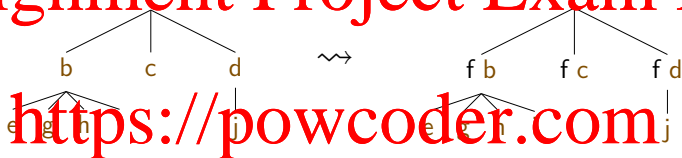
```
- : (int * bool) list data = ...
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help



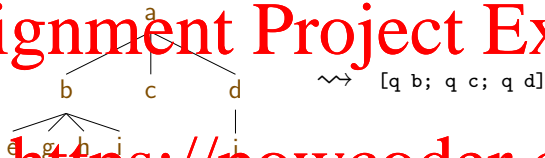
<https://powcoder.com>

Add WeChat powcoder

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

Polymorphic types for generic queries: gmapQ

Assignment Project Exam Help



<https://powcoder.com>

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

Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

```
let gmapT_int f x = x
```

```
let gmapQ_int q t = []
```

Add WeChat powcoder

Traversing datatypes: pairs

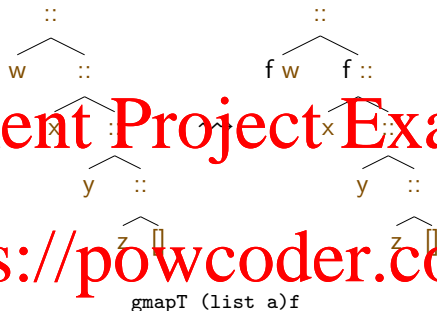
Assignment Project Exam Help

`gmap' (a * b) f`
<https://powcoder.com>

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

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

Traversing datatypes: lists



Assignment Project Exam Help

<https://powcoder.com>

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;  
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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. *)
```


Assignment Project Exam Help

<https://powcoder.com>

3. Handling recursion

Add WeChat powcoder

Generic maps, bottom up

Assignment Project Exam Help

```
let rec everywhere : generic1 -> generic1 =  
  fun f ->  
    { t =
```

```
      fun data x ->  
        if t.data  
          ((gmapT (everywhere f)) .t data x) }
```

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

Add WeChat powcoder

Generic maps, top down

Assignment Project Exam Help

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

<https://powcoder.com>

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

Add WeChat powcoder

Generic maps with a stop condition

```
let rec everywhereBut :  
  ('a -> 'b) -> ('a -> 'b) -> 'b =  
  fun stop f ->  
    { t =  
      fun data x ->  
        if stop.data x then x  
        else f data x  
      ((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 eqv t type_rep with
```

```
  | Some Ref -> g x
```

```
  | _ -> x }
```

```
everywhere
```

```
(list (bool * int))
```

```
(mkT Typeable.int succ)
```

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

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

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) }
```

<https://powcoder.com>

```
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 ->  
    { c = 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)]
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

<https://powcoder.com>

Generic printing

Add WeChat powcoder

Assignment Project Exam Help

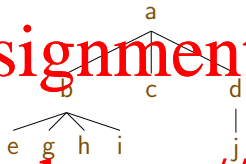
Add an additional field to data for distinguishing constructors:

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

<https://powcoder.com>
Add WeChat powcoder

A generic printing function

Assignment Project Exam Help



\rightsquigarrow "(a (b (e f) (g) (h) (d) (d i)))")"

<https://powcoder.com>

```
let rec gshow : string genericQ =
  { q =
```

```
    fun data v ->
      " " ^ data.constructor v
      ^ String.concat " " ((gmapQ gshow).q data v)
      ^ ")" }
```

Add WeChat powcoder

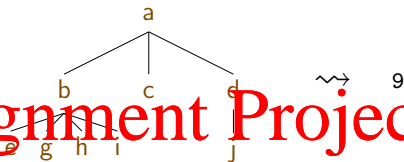
Assignment Project Exam Help

<https://powcoder.com>

Generic sizing

Add WeChat powcoder

Computing value size generically



Assignment Project Exam Help

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

```
(* gsize v -> 1 + sum ((gmapQ gsize) v) *)
```

Add WeChat powcoder

```
gsiz int 3
```

```
gsiz (list int) [1;2;3]
```

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

Remaining problems

Assignment Project Exam Help

Passing shapes around is **awkward**

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

<https://powcoder.com>

Generic traversals are **slow**

```
let gma, T_pail [ t=f ] (x, y) = (1 a x, f b y)
```

Add WeChat powcoder

Remaining problems

Assignment Project Exam Help

Passing shapes around is awkward. Solution: **implicit** everywhere

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

<https://powcoder.com>

Generic traversals are **slow**.

```
let gma, T_pail [ t=f ] (x, y) = (1 a x, f b y)
```

Add WeChat powcoder

Remaining problems

Assignment Project Exam Help

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

everywhere

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

<https://powcoder.com>

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

```
let gammaTail :: Int -> (x, y) -> (x, y)
```

Add WeChat powcoder

Next time: staging

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder