Last time: rows

$$\rho$$

This time: monads (etc.)

$$\gg\!=$$

What do monads give us?

A general approach to implementing custom effects

A reusable interface to computation

A way to structure effectful programs in a functional language

Effects

What's an effect?

An **effect** is anything a function does
besides mapping inputs to outputs.

If an expression M evaluates to a value V and changing

```
let x = M          let x = V
 in N       to      in N
```

changes the behaviour then M also performs effects.

Example effects

Effects available in OCaml	**Effects unavailable in OCaml**

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml | **Effects unavailable in OCaml**

(higher-order) state

```
r := f; !r ()
```

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml

(higher-order) state

`r := f; !r ()`

exceptions

`raise Not_found`

Effects unavailable in OCaml

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml	**Effects unavailable in OCaml**

(higher-order) state
`r := f; !r ()`
exceptions
`raise Not_found`
I/O of various sorts
`input_byte stdin`	

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml	**Effects unavailable in OCaml**

(higher-order) state

r := f; !r ()

exceptions

raise Not_found

I/O of various sorts

input_byte stdin

concurrency (interleaving)

Gc. finalise v f

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml | **Effects unavailable in OCaml**

(higher-order) state

```
r := f; !r ()
```

exceptions

```
raise Not_found
```

I/O of various sorts

```
input_byte stdin
```

concurrency (interleaving)

```
Gc.finalise v f
```

non-termination

```
let rec f x = f x
```

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml

(higher-order) state

`r := f; !r ()`

exceptions

`raise Not_found`

I/O of various sorts

`input_byte stdin`

concurrency (interleaving)

`Gc.finalise v f`

non-termination

`let rec f x = f x`

Effects unavailable in OCaml

non-determinism

`amb f g h`

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml

(higher-order) state

```
r := f; !r ()
```

exceptions

```
raise Not_found
```

I/O of various sorts

```
input_byte stdin
```

concurrency (interleaving)

```
Gc.finalise v f
```

non-termination

```
let rec f x = f x
```

Effects unavailable in OCaml

non-determinism

```
amb f g h
```

first-class continuations

```
escape x in e
```

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml

(higher-order) state

`r := f; !r ()`

exceptions

`raise Not_found`

I/O of various sorts

`input_byte stdin`

concurrency (interleaving)

`Gc.finalise v f`

non-termination

`let rec f x = f x`

Effects unavailable in OCaml

non-determinism

`amb f g h`

first-class continuations

`escape x in e`

polymorphic state

`r := "one"; r := 2`

(An **effect** is anything other than mapping inputs to outputs.)

Example effects

Effects available in OCaml

(higher-order) state

`r := f; !r ()`

exceptions

`raise Not_found`

I/O of various sorts

`input_byte stdin`

concurrency (interleaving)

`Gc.finalise v f`

non-termination

`let rec f x = f x`

Effects unavailable in OCaml

non-determinism

`amb f g h`

first-class continuations

`escape x in e`

polymorphic state

`r := "one"; r := 2`

checked exceptions

$\text{int} \xrightarrow{\text{IOError}} \text{bool}$

(An **effect** is anything other than mapping inputs to outputs.)

Capturing effects in the types

Some languages capture effects in the type system.

We might have two function arrows:

$$\text{a } \textbf{pure} \text{ arrow} \quad a \to b$$
$$\text{an } \textbf{effectful} \text{ arrow (or family of arrows)} \quad a \xrightarrow{E} b$$

and combinators for combining effectful functions

$$\begin{aligned}
\text{composeE} &: (a \xrightarrow{E} b) \to (b \xrightarrow{E} c) \to (a \xrightarrow{E} c) \\
\text{ignoreE} &: (a \xrightarrow{E} b) \to (a \xrightarrow{E} unit) \\
\text{pairE} &: (a \xrightarrow{E} b) \to (c \xrightarrow{E} d) \to (a \times c \xrightarrow{E} b \times d) \\
\text{liftPure} &: (a \to b) \to (a \xrightarrow{E} b)
\end{aligned}$$

Separating application and invocation

An alternative:

Decompose effectful arrows into functions and computations

$$a \xrightarrow{E} b \qquad \text{becomes} \qquad a \to T\ b$$

Monads

(let ... in)

Programming with monads

An imperative program

```
let id = !counter in
let () = counter := id + 1 in
  string_of_int id
```

A monadic program

```
get           >>= fun id ->
put (id + 1) >>= fun () ->
  return (string_of_int id)
```

Monads

```
module type MONAD =
sig
 type 'a t
 val return : 'a → 'a t
 val (>>=) : 'a t → ('a → 'b t) → 'b t
end
```

Monads

```
module type MONAD =
sig
 type 'a t
 val return : 'a → 'a t
 val (>>=) : 'a t → ('a → 'b t) → 'b t
end
```

Laws:

$$
\begin{aligned}
\text{return } v \gg= k &\equiv k\ v \\
v \gg= \text{return} &\equiv v \\
(m \gg= f) \gg= g &\equiv m \gg= (\text{fun } x \rightarrow f\ x \gg= g)
\end{aligned}
$$

Monad laws: intuition

Monad laws: intuition

$$\text{return } v \gg\!= k \quad \equiv \quad k\ v$$

$$\textbf{let!}\ x = v\ \textbf{in}\ M \quad \equiv \quad M[x{:=}v]$$

Monad laws: intuition

$$\text{return } v \gg\!= k \quad \equiv \quad k\ v$$

$$\textbf{let!}\ x = v\ \textbf{in}\ M \quad \equiv \quad M[x{:=}v]$$

$$v \gg\!= \text{return} \quad \equiv \quad v$$

$$\textbf{let!}\ x = M\ \textbf{in}\ x \quad \equiv \quad M$$

Monad laws: intuition

$$\text{return } v \gg\!= k \quad \equiv \quad k\ v$$

$$\textbf{let!}\ x = v \textbf{ in } M \quad \equiv \quad M[x{:=}v]$$

$$v \gg\!= \text{return} \quad \equiv \quad v$$

$$\textbf{let!}\ x = M \textbf{ in } x \quad \equiv \quad M$$

$$(m \gg\!= f) \gg\!= g \quad \equiv \quad m \gg\!= (\text{fun } x \to f\ x \gg\!= g)$$

$$\begin{array}{l}\textbf{let!}\ x = (\textbf{let!}\ y = L \textbf{ in } M) \\ \quad \textbf{in } N\end{array} \equiv \begin{array}{l}\textbf{let!}\ y = L \textbf{ in} \\ \textbf{let!}\ x = M \textbf{ in} \\ \quad N\end{array}$$

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end
```

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end

type 'a t = state → state * 'a

let return v s = (s, v)
```

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end
```

```
type 'a t = state → state * 'a
```

```
let (>>=) m k s = let s', a = m s in k a s'
```

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end

type 'a t = state → state * 'a

let get s = (s, s)
```

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end

type 'a t = state → state * 'a

let put s' _ = (s', ())
```

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end

type 'a t = state → state * 'a

let runState m ~init = m init
```

Example: a state monad

```
module type STATE = sig
  type state
  include MONAD
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end

module State (S : sig type t end)
  : STATE with type state = S.t = struct
  type state = S.t
  type 'a t = state → state * 'a
  let return v s = (s, v)
  let (>>=) m k s = let s', a = m s in k a s'
  let get s = (s, s)
  let put s' _ = (s', ())
  let runState m ~init = m init
end
```

Example: a state monad

```
type 'a tree =
    Empty : 'a tree
  | Tree : 'a tree * 'a * 'a tree → 'a tree

module IState = State (struct type t = int end)

let fresh_name : string IState.t =
  get         >>= fun i →
  put (i + 1) >>= fun () →
  return (Printf.sprintf "x%d" i)

let rec label_tree : 'a tree → string tree IState.t =
  function
    Empty → return Empty
  | Tree (l, v, r) →
    label_tree l >>= fun l →
    fresh_name   >>= fun name →
    label_tree r >>= fun r →
    return (Tree (l, name, r))
```

State satisfies the monad laws

return v >>= k

State satisfies the monad laws

$$
\begin{aligned}
& \text{return v} \gg\!= \text{k} \\
\equiv \quad & \text{(definition of return, } \gg\!= \text{)} \\
& \text{fun s} \rightarrow \text{let s', a} = (\text{fun s} \rightarrow (\text{s, v}))\ \text{s in k a s'}
\end{aligned}
$$

State satisfies the monad laws

$$
\begin{array}{ll}
& \text{return v} \gg\!= \text{k} \\
\equiv & \quad (\text{definition of return}, \gg\!=) \\
& \text{fun s} \rightarrow \text{let s', a} = (\text{fun s} \rightarrow (\text{s, v}))\ \text{s in k a s'} \\
\equiv & \quad (\beta) \\
& \text{fun s} \rightarrow \text{let s', a} = (\text{s, v})\ \text{in k a s'}
\end{array}
$$

State satisfies the monad laws

$$
\begin{array}{ll}
& \text{return v} \gg= \text{k} \\
\equiv & \quad (\text{definition of return}, \gg=) \\
& \text{fun s} \rightarrow \text{let s', a} = (\text{fun s} \rightarrow (\text{s, v}))\ \text{s in k a s'} \\
\equiv & \quad (\beta) \\
& \text{fun s} \rightarrow \text{let s', a} = (\text{s, v})\ \text{in k a s'} \\
\equiv & \quad (\beta \text{ for let}) \\
& \text{fun s} \rightarrow \text{k v s}
\end{array}
$$

State satisfies the monad laws

$$
\begin{array}{ll}
 & \text{return v} \gg\!= \text{k} \\
\equiv & (\text{definition of return}, \gg\!=) \\
 & \text{fun s} \rightarrow \text{let s', a} = (\text{fun s} \rightarrow (\text{s, v}))\ \text{s in k a s'} \\
\equiv & (\beta) \\
 & \text{fun s} \rightarrow \text{let s', a} = (\text{s, v})\ \text{in k a s'} \\
\equiv & (\beta \text{ for } \textbf{let}) \\
 & \text{fun s} \rightarrow \text{k v s} \\
\equiv & (\eta) \\
 & \text{k v}
\end{array}
$$

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(error → 'a) → 'a
end
```

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(error → 'a) → 'a
end

type 'a t =
    Val : 'a → 'a t
  | Exn : error → 'a t

let return v = Val v
```

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(error → 'a) → 'a
end

type 'a t =
    Val : 'a → 'a t
  | Exn : error → 'a t

let (>>=) m k = match m with
  Val v → k v | Exn e → Exn e
```

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(error → 'a) → 'a
end

type 'a t =
    Val : 'a → 'a t
  | Exn : error → 'a t

let raise e = Exn e
```

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(error → 'a) → 'a
end

type 'a t =
   Val : 'a → 'a t
 | Exn : error → 'a t

let _try_ m ~catch = match m with
  Val v → v | Exn e → catch e
```

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(error → 'a) → 'a
end

module Error (E: sig type t end)
  : ERROR with type error = E.t = struct
  type error = E.t
  type 'a t =
      Val : 'a → 'a t
    | Exn : error → 'a t
  let return v = Val v
  let (>>=) m k = match m with
    Val v → k v | Exn e → Exn e
  let raise e = Exn e
  let _try_ m ~catch = match m with
    Val v → v | Exn e → catch e
end
```

Example: exception

```
let rec mapMTree f = function
    Empty → return Empty
  | Tree (l, v, r) →
    mapMTree f l >>= fun l →
    f v          >>= fun v →
    mapMTree f r >>= fun r →
    return (Tree (l, v, r))

let check_nonzero =
  mapMTree
    (fun v →
        if v = 0 then raise Zero
        else return v)
```

Exception satisfies the monad laws

$v \gg= \text{return}$

Exception satisfies the monad laws

$$\begin{aligned} & \mathsf{v} \gg\!= \mathsf{return} \\ \equiv \quad & (\text{definition of return}, \gg\!=) \\ & \mathsf{match\ v\ with\ Val\ v} \rightarrow \mathsf{Val\ v} \mid \mathsf{Exn\ e} \rightarrow \mathsf{Exn\ e} \end{aligned}$$

Exception satisfies the monad laws

$$
\begin{aligned}
& \mathsf{v} \gg= \mathsf{return} \\
\equiv \quad & (\text{definition of return}, \gg=) \\
& \mathsf{match\ v\ with\ Val\ v} \to \mathsf{Val\ v} \mid \mathsf{Exn\ e} \to \mathsf{Exn\ e} \\
\equiv \quad & (\eta \text{ for sums}) \\
& \mathsf{v}
\end{aligned}
$$

Indexed monads

$(\Gamma \vdash M : A \,!\, e)$

Indexed monads and effect systems

A computation of type `('e, 'a) t`
performs an effect `'e`
produces a result of type `'a`.

Strengthening the interface: indexed monads

```
module type INDEXED_MONAD =
sig
  type ('e, 'a) t
  val return : 'a → (_, 'a) t
  val (>>=) : ('e,'a) t →
        ('a → ('e,'b) t) →
                ('e,'b) t
end
```

(Laws: as for monads.)

An indexed monad for exceptions

```
module type IERROR =
sig
  include INDEXED_MONAD
  val raise : 'e → ('e, _) t
  val _try_ : ('e, 'a) t →
        catch:('e → 'a) →
              'a
end
```

An indexed monad for exceptions

```
module IError : IERROR =
struct
  type ('e, 'a) t =
      Val : 'a → ('e, 'a) t
    | Exn : 'e → ('e, 'a) t
  let return v = Val v
  let raise e = Exn e
  let (>>=) m k =
    match m with
      Val v → k v
    | Exn e → Exn e
  let _try_ m ~catch =
    match m with
      Val v → v
    | Exn e → catch e
end
```

Indexed monads and rows

```
let rec find p = function
  [] → raise `Not_found
| x :: _ when p x → return x
| _ :: xs → find p xs

let pop = function
  [] → raise (`Empty "pop")
| x :: xs → return (x, xs)

let gt_0 x = x > 0

pop []         >>= fun (_, xs) →
find gt_0 xs >>= fun y →
return y
```

```
let rec find p = function
  [] → raise Not_found
| x :: _ when p x → x
| _ :: xs → find p xs

let pop = function
  [] → raise (Empty "pop")
| x :: xs → (x, xs)

let gt_0 x = x > 0

let _, xs = pop [] in
let y = find gt_0 xs in
  y
```

Parameterised monads

({P} C {Q})

Parameterised monads and Hoare Logic

A computation of type `('p, 'q, 'a) t`
has *precondition* `'p`
has *postcondition* `'q`
produces a result of type `'a`.

i.e. `('p, 'q, 'a) t` is a kind of Hoare triple $\{P\}$ M $\{Q\}$.

Strengthening the interface: parameterised monads

```
module type PARAMETERISED_MONAD =
sig
  type ('s,'t,'a) t
  val return : 'a → ('s,'s,'a) t
  val (>>=) : ('r,'s,'a) t →
          ('a → ('s,'t,'b) t) →
                  ('r,'t,'b) t
end
```

(Laws: as for monads.)

A parameterised monad for state

```
module type PSTATE =
sig
 include PARAMETERISED_MONAD
 val get : ('s,'s,'s) t
 val put : 's → (_,'s,unit) t
 val runState : ('s,'t,'a) t → init:'s → 't * 'a
end
```

A parameterised monad for state

```
module PState : PSTATE =
struct
  type ('s, 't, 'a) t = 's → 't * 'a
  let return v s = (s, v)
  let (>>=) m k s = let t, a = m s in k a t
  let put s _ = (s, ())
  let get s = (s, s)
  let runState m ~init = m init
end
```

Programming with polymorphic state

```
type (_, _) instr =
   Add : (int * (int * 's),
                  int * 's) instr
 | If  : (bool * ('a * ('a * 's)),
                       'a * 's) instr
 | PushConst : 'a → ('s,
                  'a * 's) instr

type (_, _) instrs =
   Stop : ('s, 's) instrs
 | ::   : ('s1, 's2) instr
        * ('s2, 's3) instrs →
          ('s1, 's3) instrs

let program =
  PushConst 3 :: PushConst 4 :: PushConst 5 ::
  PushConst true :: If :: Add :: Stop
```

Programming with polymorphic state

```
let add (x,(y,s)) = (x+y,s)
let _if_ (c,(t,(e,s))) = ((if c then t else e),s)
let push_const k s = (k, s)

let applyS f = get >>= fun s → put (f s)

let exec1 : type a b.(a,b) instr → (a,b,unit) Pstate.t =
  function
     Add → applyS add
   | If → applyS _if_
   | PushConst k → applyS (push_const k)

let rec exec
 : type a b.int → (a,b) instrs → (a,b,int) Pstate.t =
  fun c → function
      i :: is → exec1 i >>= fun () →
                  exec (succ c) is
    | Stop → return c
```

Next time:

The struggle for power