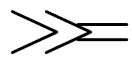
Last time: Overloading

```
val (=) : {E:EQ} → E.t → E.t → bool
```

This time: monads (etc.)



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
 in N
```

to

```
let x = V
 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

(higher-order) state

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

Effects unavailable in OCaml

(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

(higher-order) state

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

exceptions

```
raise Not_found
```

I/O of various sorts

```
input_byte stdin
```

Effects unavailable in OCaml

(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
```

Effects unavailable in OCaml

(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

(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

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

(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:

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

and combinators for combining effectful functions

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

Separating application and performing effects

Alternative approach

Decompose effectful arrows into pure functions and computations

$$a \rightsquigarrow b \qquad \text{becomes} \qquad a \to T\ b$$

Monads

(`let x = e 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

let return {M:MONAD} x = M.return x
let (>>=) {M:MONAD} m k = M.(>>=) m k
```

Monads

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

let return {M:MONAD} x = M.return x
let (>>=) {M:MONAD} m k = M.(>>=) m k
```

Laws:

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

Monad laws: intuition

Monad laws: intuition

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

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

Monad laws: intuition

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

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

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

$$\texttt{let x = M in x} \quad \equiv \quad \texttt{M}$$

Monad laws: intuition

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

```
let x = v in M    ≡    M[x:=v]
```

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

```
let x = M in x    ≡    M
```

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

```
let x = (let y = L in M)        ≡    let y = L in
   in N                              let x = M in
                                        N
```

Example: a state monad

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

implicit module Monad_of_state{S:STATE} = S.Monad
```

Example: a state monad

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

type 'a t = state → state * 'a

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

Example: a state monad

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

module State (S : sig type t end) = struct
  type state = S.t
  type 'a t = state -> state * 'a
  module Monad = struct
    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'
  end
  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

implicit 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 $\gg\!=$ 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

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

State satisfies the monad laws

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

State satisfies the monad laws

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

Example: exception

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


let rec find : 'a. ('a → bool) → 'a list → 'a t =
  fun p l → match l with
       [] → raise "Not found!"
    | x :: _ when p x → return x
    | _ :: xs → find p xs


  _try_ (
    find (greater 3) l >>= fun v →
    return (string_of_int v)
  )
  (fun error → error)
```

Example: exception

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

module Error (E: sig type t end) = struct
  type error = E.t
  module Monad = struct
    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
  end
  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 : 'a. {M:MONAD} → ('a → 'b M.t) → '
    a tree → 'b tree M.t =
  fun {M:MONAD} f l → match l with
    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

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

Exception satisfies the monad laws

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

Higher-order effectful programs

Monadic effects are higher-order

$$
\begin{aligned}
\mathsf{composeE} &: (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c) \\
\mathsf{pairE} &: (a \rightsquigarrow b) \to (c \rightsquigarrow d) \to (a \times c \rightsquigarrow b \times d) \\
\mathsf{uncurryE} &: (a \rightsquigarrow b \rightsquigarrow c) \to (a \times b \rightsquigarrow c) \\
\mathsf{liftPure} &: (a \to b) \to (a \rightsquigarrow b)
\end{aligned}
$$

Higher-order computations with monads

```
val composeM : {M:MONAD} →
 ('a → 'b M.t) → ('b → 'c M.t) → ('a → 'c M.t)
```

```
let composeM {M:MONAD} f g x : _ M.t =
    f x >>= fun y →
    g y
```

```
val uncurryM : {M:MONAD} →
 ('a → ('b → 'c M.t) M.t) → (('a * 'b) → 'c M.t)
```

```
let uncurryM {M:MONAD} f (x,y) : _ M.t =
    f x >>= fun g →
    g y
```

Applicatives

(`let x = e ... and`)

Allowing only "static" effects

Idea: stop information flowing from one computation into another.

Only allow unparameterised computations:

$$1 \rightsquigarrow b$$

We can no longer write functions like this:

$$\mathsf{composeE} \quad : \quad (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c)$$

but some useful functions are still possible:

$$\mathtt{pairE}_{\mathtt{static}} \ : \ (1 \rightsquigarrow a) \to (1 \rightsquigarrow b) \to (1 \rightsquigarrow a \times b)$$

Applicative programs

An imperative program

```
let x = fresh_name ()
and y = fresh_name ()
 in (x, y)
```

An applicative program

```
    pure (fun x y → (x, y))
⊗ fresh_name
⊗ fresh_name
```