Last time: GADTs

$$a \equiv b$$

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

(higher-order) state

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

exceptions

```
raise Not_found
```

(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 **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

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

(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:

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

and combinators for combining effectful functions

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

Separating application and performing effects

An alternative:

Decompose effectful arrows into 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
```

Monads

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

Laws:

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

Monad laws: intuition

Monad laws: intuition

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

`let x = v in M` $\equiv$ `M[x:=v]`

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
```

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
```

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

```
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
  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

$\text{return } v \gg= k$

State satisfies the monad laws

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

State satisfies the monad laws

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

State satisfies the monad laws

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

State satisfies the monad laws

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

Example: exception

```
module type ERROR = sig
  type error
  include MONAD
  val raise : error → 'a t
  val _try_ : 'a t → catch:(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 ~than:3) l >>= fun v →
    return (string_of_int v)
  ) ~catch:(fun error → error)
```

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

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

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.)

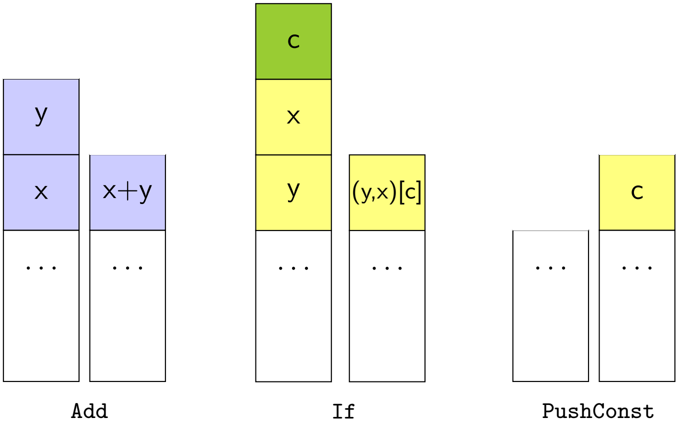
A parameterised monad for state

```
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

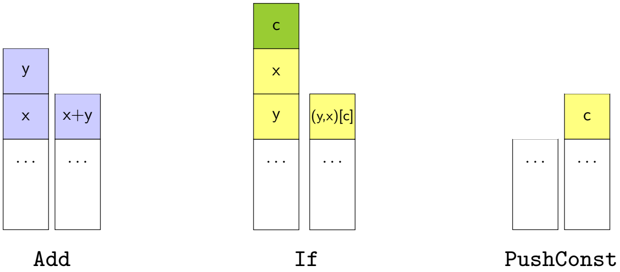
```
module PState : PSTATE =
struct
  type ('s, 't, 'a) t = 's → 't * 'a
  let return v s = (s, v)
  let (>>=) m k s = let t, a = m s in k a t
  let put s _ = (s, ())
  let get s = (s, s)
  let runState m ~init = m init
end
```

Programming with polymorphic state: a stack machine



Programming with polymorphic state: a stack machine

```
module type STACK_OPS =
sig
  type ('s,'t,'a) t
  val add : (int * (int * 's),
                  int * 's, unit) t
  val _if_ : (bool * ('a * ('a * 's)),
                        'a * 's, unit) t
  val push_const : 'a → ('s,
                   'a * 's, unit) t
end
```



Programming with polymorphic state

```
module type STACKM = sig
 include PARAMETERISED_MONAD
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t → 's → 't * 'a
end

module StackM : STACKM = struct
 include PState

 let add = get >>= fun (x,(y,s)) → put (x+y,s)
 let _if_ = get  >>= fun (c,(t,(e,s))) →
    put (if c then t else e, s)
 let push_const k = get >>= fun s → put (k, s)
 let execute = runState
end
```

Higher-order effectful programs

Monadic effects are higher-order

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

Higher-order effects with monads

```
val composeM :
 ('a → 'b t) → ('b → 'c t) → ('a → 'c t)
```

```
let composeM f g x =
   f x >>= fun y →
   g y
```

```
val uncurryM :
 ('a → ('b → 'c t) t) → (('a * 'b) → 'c t)
```

```
let uncurryM f (x,y) =
   f x >>= fun g →
   g y
```

Next time: arrows, applicatives (etc.)

$$\ggg$$

$$<*>$$