Last time: generic programming

```
val (=) : {D:DATA} → D.t → D.t → bool
```

This time: monads etc., continued

$\gg=$ $\otimes$ `effect E`

Recap: monads, bind and let

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)
```

Recap: 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
```

Applicatives

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

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
```

Applicatives

```
module type APPLICATIVE =
sig
 type 'a t
 val pure : 'a → 'a t
 val (⊗) : ('a → 'b) t → 'a t → 'b t
end

let pure {A:APPLICATIVE} x = A.pure x
let (⊗) {A:APPLICATIVE} m k = A.(⊗) m k
```

Applicatives

```
module type APPLICATIVE =
sig
 type 'a t
 val pure : 'a → 'a t
 val (⊗) : ('a → 'b) t → 'a t → 'b t
end

let pure {A:APPLICATIVE} x = A.pure x
let (⊗) {A:APPLICATIVE} m k = A.(⊗) m k
```

Laws:

$$\begin{array}{rcl} \text{pure f} \otimes \text{pure v} & \equiv & \text{pure (f v)} \\ \text{u} & \equiv & \text{pure id} \otimes \text{u} \\ \text{u} \otimes (\text{v} \otimes \text{w}) & \equiv & \text{pure compose} \otimes \text{u} \otimes \text{v} \otimes \text{w} \\ \text{v} \otimes \text{pure x} & \equiv & \text{pure (fun f} \rightarrow \text{f x)} \otimes \text{v} \end{array}$$

$\gg=$ vs $\otimes$

The type of $\gg=$: `'a t → ('a → 'b t) → 'b t`

`'a → 'b t`: a function that builds a computation

(Almost) the type of $\otimes$: `'a t → ('a → 'b) t → 'b t`

`('a → 'b) t`: a computation that builds a function

The actual type of $\otimes$: `('a → 'b) t → 'a t → 'b t`

Applicative normal forms

```
pure f ⊗ c₁ ⊗ c₂ ... ⊗ cₙ
```

```
pure (fun x₁ x₂ ... xₙ → e) ⊗ c₁ ⊗ c₂ ... ⊗ cₙ
```

```
let x₁ = c₁
and x₂ = c₂
 ...
and xₙ = cₙ
  in e
```

Applicative normalisation via the laws

pure f $\otimes$ (pure g $\otimes$ `fresh_name`) $\otimes$ `fresh_name`

Applicative normalisation via the laws

$$\begin{array}{ll} & \mathsf{pure}\ \mathsf{f} \otimes (\mathsf{pure}\ \mathsf{g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \\ \equiv & (\text{composition law}) \\ & (\mathsf{pure}\ \mathsf{compose} \otimes \mathsf{pure}\ \mathsf{f} \otimes \mathsf{pure}\ \mathsf{g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \end{array}$$

Applicative normalisation via the laws

$$
\begin{array}{ll}
 & \text{pure f} \otimes (\text{pure g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \\
\equiv & (\text{composition law}) \\
 & (\text{pure compose} \otimes \text{pure f} \otimes \text{pure g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \\
\equiv & (\text{homomorphism law } (\times 2)) \\
 & \text{pure (compose f g)} \otimes \texttt{fresh\_name} \otimes \texttt{fresh\_name}
\end{array}
$$

Creating applicatives: every monad is an applicative

```
implicit module Applicative_of_monad {M:MONAD} :
  APPLICATIVE with type 'a t = 'a M.t =
struct
  type 'a t = 'a M.t
  let pure = M.return
  let (⊗) f p =
     M.(f >>= fun g →
        p >>= fun q →
        return (g q))
end
```

The state applicative via the state monad

```
module StateA(S : sig type t end) :
sig
  type state = S.t
  type ’a t
  module Applicative : APPLICATIVE with type ’a t = ’a t
  val get : state t
  val put : state → unit t
  val runState : ’a t → state → state * ’a
end =
struct
  type state = S.t
  module StateM = State(S)
  type ’a t = ’a StateM.t
  module Applicative =
     Applicative_of_monad{StateM.Monad}
  let (get, put, runState) = StateM.(get, put, runState)
end
```

Creating applicatives: composing applicatives

```
module Compose (F : APPLICATIVE)
               (G : APPLICATIVE) :
  APPLICATIVE with type 'a t = 'a G.t F.t =
struct
  type 'a t = 'a G.t F.t
  let pure x = F.pure (G.pure x)
  let (⊗) f x = F.(pure G.(⊗) ⊗ f ⊗ x)
end
```

Creating applicatives: the dual applicative

```
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
```

```
module RevNameA = Dual_applicative(NameA.Applicative)

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

Composed applicatives are law-abiding

pure f $\otimes$ pure x

Composed applicatives are law-abiding

$$
\begin{array}{ll}
 & \text{pure f} \otimes \text{pure x} \\
\equiv & \quad \text{(definition of } \otimes \text{ and pure)} \\
 & \text{F.pure } (\otimes_G\ ) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)}
\end{array}
$$

Composed applicatives are law-abiding

$$
\begin{array}{ll}
 & \text{pure f} \otimes \text{pure x} \\
\equiv & \quad \text{(definition of } \otimes \text{ and pure)} \\
 & \text{F.pure } (\otimes_G\ ) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv & \quad \text{(homomorphism law for F } (\times 2)) \\
 & \text{F.pure (G.pure f} \otimes_G \text{G.pure x)}
\end{array}
$$

Composed applicatives are law-abiding

$$
\begin{aligned}
& \text{pure f} \otimes \text{pure x} \\
\equiv \quad & (\text{definition of } \otimes \text{ and pure}) \\
& \text{F.pure } (\otimes_G\ ) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv \quad & (\text{homomorphism law for F } (\times 2)) \\
& \text{F.pure (G.pure f } \otimes_G \text{ G.pure x)} \\
\equiv \quad & (\text{homomorphism law for G}) \\
& \text{F.pure (G.pure (f x))}
\end{aligned}
$$

Composed applicatives are law-abiding

$$
\begin{aligned}
& \text{pure f} \otimes \text{pure x} \\
\equiv \quad & (\text{definition of } \otimes \text{ and pure}) \\
& \text{F.pure } (\otimes_G\ ) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv \quad & (\text{homomorphism law for F } (\times 2)) \\
& \text{F.pure (G.pure f} \otimes_G \text{G.pure x)} \\
\equiv \quad & (\text{homomorphism law for G}) \\
& \text{F.pure (G.pure (f x))} \\
\equiv \quad & (\text{definition of pure}) \\
& \text{pure (f x)}
\end{aligned}
$$

Fresh names, monadically

```
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))
```

Naming as a primitive effect

Problem: we cannot write `fresh_name` using the `APPLICATIVE` interface.

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

Solution: introduce `fresh_name` as a primitive effect:

```
implicit module NameA : sig
  module Applicative : APPLICATIVE
  val fresh_name : string Applicative.t
end = ...
```

Traversing with namer

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

The phantom monoid applicative

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end

module Phantom_monoid (M: MONOID)
  : APPLICATIVE with type 'a t = M.t =
struct
  type 'a t = M.t
  let pure _ = M.zero
  let (⊗) = M.(++)
end
```

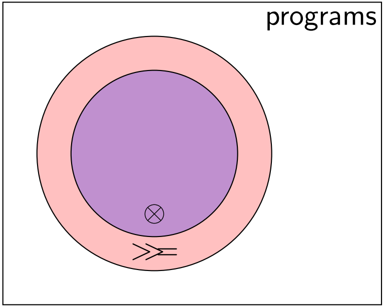
The phantom monoid applicative

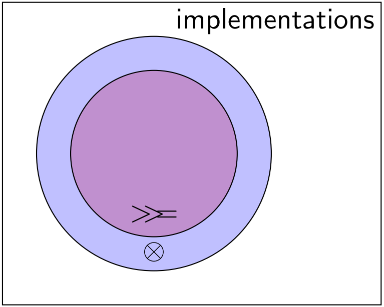
```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end

module Phantom_monoid (M: MONOID)
  : APPLICATIVE with type 'a t = M.t =
struct
  type 'a t = M.t
  let pure _ = M.zero
  let (⊗) = M.(++)
end
```

Observation: we cannot implement `Phantom_monoid` as a monad.

Applicatives vs monads





Some monadic programs are not applicative, e.g. `fresh_name`.

Some applicative instances are not monadic, e.g. `Phantom_monoid`.

Guideline: Postel's law

Be conservative in what you do,
be liberal in what you accept from others.

Guideline: Postel's law

Be conservative in what you do,
be liberal in what you accept from others.

Conservative in what you do: **use** applicatives, not monads.
(Applicatives give the implementor more freedom.)

Guideline: Postel's law

Be conservative in what you do,
be liberal in what you accept from others.

Conservative in what you do: **use** applicatives, not monads.
(Applicatives give the implementor more freedom.)

Liberal in what you accept: **implement** monads, not applicatives.
(Monads give the user more power.)

Summary

monads

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

applicatives

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

Algebraic effects and handlers

(`effect E`)

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x → g x
| B y → h y
| ...
```

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x → g x
| B y → h y
| ...
```

`f v` *evaluates to* `A x`: *evaluate* `g x`

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x → g x
| B y → h y
| ...
```

`f v` *evaluates to* `A x`: *evaluate* `g x`

`f v` *evaluates to* `B y`: *evaluate* `h y`

...

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x → g x
| B y → h y
| ...
```

`f v` *evaluates to* `A x`: *evaluate* `g x`

`f v` *evaluates to* `B y`: *evaluate* `h y`

...

`f v` *raises an* **exception** `E`: *raise* `E`

Extending `match` for exceptions

Write:

```
match f v with
| A x → g x
| B y → h y
| ...
| exception (E z) → j z
```

E.g. search an association list `l` (type `(string * bool) list`):

```
match List.assoc s l with
| true →  "found (True)"
| false → "found (False)"
| exception Not_found → "not found"
```

Extending `match` for effects

Possible outcomes of `match`

```
match f v with
| A x → g x
| B y → h y
| exception (E z) → j z
| ...
```

`f v` *evaluates to* `A x`: *evaluate* `g x`

`f v` *evaluates to* `B y`: *evaluate* `h y`

`f v` *raises an* **exception** `E`: *raise* `E`

...

Extending `match` for effects

Possible outcomes of `match`

```
match f v with
| A x → g x
| B y → h y
| exception (E z) → j z
| ...
```

`f v` *evaluates to* `A x`: *evaluate* `g x`

`f v` *evaluates to* `B y`: *evaluate* `h y`

`f v` *raises an* **exception** `E`: *raise* `E`

...

`f v` *performs an* **effect** *E and continues*: *perform* `E` and continue

Elements of exceptions

Exceptions

`exception E: s → exn` (*means* `type exn += E: s → exn`)

Raising exceptions

```
val raise : exn → 'b
```

Handling exceptions

```
match e with
...
| exception (E x) → ...
```

Elements of effects

Effects

`effect E: s → t` (*means* `type _ eff += E: s → t eff`)

Performing effects

```
val perform : 'a eff → 'a
```

Handling effects

```
match e with
...
| effect (E x) k → ...
```

Running continuations

```
val continue : ('a, 'b) continuation → 'a → 'b
```

Using effects: yet another ocaml fork

modular implicits

```
opam switch 4.02.0+modular-implicits
```

effects

```
opam remote add advanced-fp \
  git://github.com/ocamllabs/advanced-fp-repo
opam switch 4.03.0+effects
```

staging (next week)

```
opam switch 4.02.1+modular-implicits-ber
```

Example: exceptions as an effect

Define the effect and a function to perform the effect:

```
effect Raise : exn → 'a
let raise e = perform (Raise e)
```

Define a function to handle the effect:

```
let _try_ f handle =
 match f () with
 | v → v
 | effect (Raise e) k → (* discard k! *) handle e
```

Program in direct (non-monadic) style:

```
let rec assoc x = function
| [] → raise Not_found
| (k,v)::t → if k = x then v else assoc x t

_try_ (fun () → Some (assoc 3 l))
      (fun ex → None)
```

Recap: state as a monad

The type of computations:

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

The `return` and ≫= functions from `MONAD`:

```
let return v s = (s, v)
let (≫=) m k s = let s', a = m s in k a s'
```

Signatures of primitive effects:

```
val get : state t
val put : state → unit t
```

Primitive effects and a *run* function:

```
let get s = (s, s)
let put s' _ = (s', ())
let runState m init = m init
```

Example: state as an effect

Primitive effects:

```
effect Put : state → unit
effect Get : state
```

Functions to perform effects:

```
let put v = perform (Put v)
let get () = perform Get
```

A handler function:

```
let run f init =
  let exec =
    match f () with
    | x → (fun s → (s, x))
    | effect (Put s') k → (fun s → continue k () s')
    | effect Get k → (fun s → continue k s s)
  in exec init
```

Evaluating an effectful program

```
let run f init =
  let exec =
    match f () with
    | x → (fun s → (s, x))
    | effect (Put s’) k → (fun s → continue k () s’)
    | effect Get k → (fun s → continue k s s)
  in exec init
```

```
run (fun () →
  let id = get () in
  let () = put (id + 1) in
    string_of_int id
) 3
```

Evaluating an effectful program

```
(match (fun () →
         let id = get () in
         let () = put (id + 1) in
           string_of_int id) ()
 with
 | x → (fun s → (s, x))
 | effect (Put s’) k → (fun s → continue k () s’)
 | effect Get k → (fun s → continue k s s))
 3
```

Evaluating an effectful program

```
(match (let id = get () in
        let () = put (id + 1) in
          string_of_int id)
 with
 | x → (fun s → (s, x))
 | effect (Put s’) k → (fun s → continue k () s’)
 | effect Get k → (fun s → continue k s s))
 3
```

Evaluating an effectful program

```
(match (let id = perform Get in
        let () = put (id + 1) in
          string_of_int id)
 with
 | x → (fun s → (s, x))
 | effect (Put s’) k → (fun s → continue k () s’)
 | effect Get k → (fun s → continue k s s))
 3
```

Evaluating an effectful program

```
(fun s → continue k s s) 3
```

Evaluating an effectful program

```
continue k 3 3

k =
 (match (let id = - in
         let () = put (id + 1) in
           string_of_int id)
  with
  | x → (fun s → (s, x))
  | effect (Put s') k → (fun s → continue k () s')
  | effect Get k → (fun s → continue k s s))
```

Evaluating an effectful program

```
(match (let id = 3 in
        let () = put (id + 1) in
          string_of_int id)
 with
 | x → (fun s → (s, x))
 | effect (Put s’) k → (fun s → continue k () s’)
 | effect Get k → (fun s → continue k s s)) 3
```

Evaluating an effectful program

```
(match (let () = put (3 + 1) in
          string_of_int 3)
 with
 | x → (fun s → (s, x))
 | effect (Put s') k → (fun s → continue k () s')
 | effect Get k → (fun s → continue k s s)) 3
```

Evaluating an effectful program

```
(match (let () = perform (Put 4) in
          string_of_int 3)
 with
 | x → (fun s → (s, x))
 | effect (Put s’) k → (fun s → continue k () s’)
 | effect Get k → (fun s → continue k s s)) 3
```

Evaluating an effectful program

```
 (fun s → continue k () 4) 3

k =
 (match (let () = - in
            string_of_int 3)
  with
  | x → (fun s → (s, x))
  | effect (Put s') k → (fun s → continue k () s')
  | effect Get k → (fun s → continue k s s))
```

Evaluating an effectful program

```
(match (let () = () in
          string_of_int 3)
 with
 | x → (fun s → (s, x))
 | effect (Put s') k → (fun s → continue k () s')
 | effect Get k → (fun s → continue k s s))
4
```

Evaluating an effectful program

```
(match string_of_int 3
 with
 | x → (fun s → (s, x))
 | effect (Put s') k → (fun s → continue k () s')
 | effect Get k → (fun s → continue k s s))
4
```

Evaluating an effectful program

```
(match "3"
 with
 | x → (fun s → (s, x))
 | effect (Put s') k → (fun s → continue k () s')
 | effect Get k → (fun s → continue k s s))
4
```

Evaluating an effectful program

```
(fun s → (s, "3")) 4
```

Evaluating an effectful program

```
(4, "3")
```

Effects and monads

Integrating effects and monads

What we'll get

Easy reuse of existing monadic code

(Uniformly turn monads into effects)

Improved efficiency, eliminating unnecessary binds

(Normalize computations before running them)

No need to write in monadic style

Use `let` instead of $\gg=$

“Unnecessary” binds

The monad laws tell us that the following are equivalent:

$$\begin{aligned} \text{return v} \gg\!= \text{k} &\quad\equiv\quad \text{k v} \\ \text{v} \gg\!= \text{return} &\quad\equiv\quad \text{v} \end{aligned}$$

Why would we ever write the lhs?

“Administrative” $\gg\!=$ and `return` arise through **abstraction**

```
let apply f x = f >>= fun g →
                x >>= fun y →
                return (g y)
...
apply (return succ) y
(* used: two returns, two ≫=s *)
(* needed: one return, one ≫= *)
```

Effects from monads: the elements

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

Given `M : MONAD`:

```
  effect E : 'a M.t → 'a
```

```
  let reify f = match f () with
    | x → M.return x
    | effect (E m) k → M.bind m (continue k)
```

```
  let reflect m = perform (E m)
```

Effects from monads: the functor

```
module RR(M: MONAD) :
sig
  val reify : (unit → 'a) → 'a M.t
  val reflect : 'a M.t → 'a
end =
struct
  effect E : 'a M.t → 'a

  let reify f = match f () with
    | x → M.return x
    | effect (E m) k → M.bind m (continue k)

  let reflect m = perform (E m)
end
```

Example: state effect from the state monad

```
module StateR = RR(State)
```

Build effectful functions from primitive effects `get`, `put`:

```
module StateR = RR(State)
let put v = StateR.reflect (State.put v)
let get () = StateR.reflect State.get
```

Build the handler from `reify` and `State.run`:

```
let run_state f init = State.run (StateR.reify f) init
```

Use `let` instead of $\gg=$:

```
let id = get () in
let () = put (id + 1) in
  string_of_int id
```

Summary

Applicatives are a weaker, more general interface to effects
(⊗ is less powerful than $\gg\!=$)

Every applicative program can be written with monads
(but not vice versa)

Every `Monad` instance has a corresponding `Applicative` instance
(but not vice versa)

We can build effects using handlers

Existing monads transfer uniformly

Next time: multi-stage programming

`.< e >.`