Monads
(and applicatives)

February 2018

Last time: Overloading

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

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.

Rough guideline

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.)

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

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

resumable exceptions

`(invoke-restart "Try again")`

(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:

a **pure** arrow $a \to b$

an **effectful** arrow (or family of arrows) $a \leadsto b$

and combinators for combining effectful functions

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

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 ...`)

Plan: define a `let`-like interface, then define its behaviour.

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)
```

```
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
```

```
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

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

Monad laws: intuition

The **left identity** is a kind of β law:

```
return v >>= k   ≡   k v
let x = v in M   ≡   M[x:=v]
```

The **right identity** is a kind of η law:

```
   e >>= return   ≡   e
 let x = M in x   ≡   M
```

The **associativity law** is a kind of commuting conversion:

```
(e >>= f) >>= g   ≡   e >>= (fun x → f x >>= g)
```

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

Example: a state monad (interface)

The `STATE` interface extends `MONAD` with two effects, `get` and `put`:

```
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
```

Every instance of `STATE` can be used as a `MONAD`:

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

Example: a state monad (implementation)

```
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
```

A computation in `STATE` transforms a `state` and returns a result `'a`:

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

`return` builds a computation that leaves the state untouched:

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

Example: a state monad (implementation)

```
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
```

A computation in `STATE` transforms a `state` and returns a result `'a`:

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

$\gg=$ threads the state from one computation to another:

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

Example: a state monad (implementation)

```
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
```

A computation in `STATE` transforms a `state` and returns a result `'a`:

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

`get` returns the current state (and leaves the state untouched):

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

Example: a state monad (implementation)

```
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
```

A computation in `STATE` transforms a `state` and returns a result `'a`:

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

`put` replaces the current state:

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

Example: a state monad (implementation)

```
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
```

A computation in `STATE` transforms a `state` and returns a result `'a`:

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

`runState` runs a computation with an initial state:

```
  let runState m init = m init
```

Example: a state monad (implementation)

```
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 (use)

```
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 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 label_tree = mapMTree (fun _ → fresh_name)
```

State satisfies the monad laws

Example: we'll prove the following law for the state monad:

$$\texttt{return v} \gg= \texttt{k} \equiv \texttt{k v}$$

```
return v >>= k
```

State satisfies the monad laws

Example: we'll prove the following law for the state monad:

$$\texttt{return v} \gg= \texttt{k} \equiv \texttt{k v}$$

```
  return v >>= k
≡   (definition of return, >>=)
  fun s→ let s', a = (fun s→ (s, v)) s in k a s'
```

State satisfies the monad laws

Example: we'll prove the following law for the state monad:

$$\texttt{return v} \gg\!= \texttt{k} \equiv \texttt{k v}$$

```
  return v >>= k
≡    (definition of return, >>=)
  fun s→ let s', a = (fun s→ (s, v)) s in k a s'
≡    (β)
  fun s→ let s', a = (s, v) in k a s'
```

State satisfies the monad laws

Example: we'll prove the following law for the state monad:

$$\texttt{return v} \gg\!= \texttt{k} \equiv \texttt{k v}$$

```
  return v >>= k
≡   (definition of return, >>=)
  fun s→ let s', a = (fun s→ (s, v)) s in k a s'
≡   (β)
  fun s→ let s', a = (s, v) in k a s'
≡   (β for let)
  fun s→ k v s
```

State satisfies the monad laws

Example: we'll prove the following law for the state monad:

$$\texttt{return v} \gg\!= \texttt{k} \equiv \texttt{k v}$$

```
  return v >>= k
≡    (definition of return, >>=)
  fun s→ let s', a = (fun s→ (s, v)) s in k a s'
≡    (β)
  fun s→ let s', a = (s, v) in k a s'
≡    (β for let)
  fun s→ k v s
≡    (η)
  k 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
```

Using the error monad:

```
  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
```

Running an error computation:

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

Example: exception (implementation)

```
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
```

A computation in `ERROR` is either a result of type `'a` or an error:

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

`return` builds a successful computation (i.e. a result):

```
  let return v = Val v
```

Example: exception (implementation)

```
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
```

A computation in `ERROR` is either a result of type `'a` or an error:

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

≫= runs the second computation only if the first succeeds:

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

Example: exception (implementation)

```
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
```

A computation in `ERROR` is either a result of type `'a` or an error:

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

`raise` builds a failed computation (i.e. an error)

```
  let raise e = Exn e
```

Example: exception (implementation)

```
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
```

A computation in `ERROR` is either a result of type `'a` or an error:

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

`_try_` runs a computation and handles the two possible outcomes:

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

Example: exception (implementation)

```
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 (use)

```
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

Example: we'll prove the following law for the exception monad:

$$\texttt{v} \gg= \texttt{return} \equiv \texttt{v}$$

$\texttt{v} \gg= \texttt{return}$

Exception satisfies the monad laws

Example: we'll prove the following law for the exception monad:

$$\texttt{v} \gg= \texttt{return} \equiv \texttt{v}$$

$\texttt{v} \gg= \texttt{return}$

$\equiv$ (definition of `return`, $\gg=$)

`match v with Val v → Val v | Exn e → Exn e`

Exception satisfies the monad laws

Example: we'll prove the following law for the exception monad:

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

$\texttt{v} \gg\!= \texttt{return}$

$\equiv$ (definition of `return`, $\gg\!=$)

`match v with Val v → Val v | Exn e → Exn e`

$\equiv$ (η for sums)

`v`

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 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:

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

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
```

```
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
```

```
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:

```
pure f ⊗ pure v  ≡  pure (f v)
              u  ≡  pure id ⊗ u
  u ⊗ (v ⊗ w)    ≡  pure compose ⊗ u ⊗ v ⊗ w
   v ⊗ pure x    ≡  pure (fun f→ f x) ⊗ v
```

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

Every applicative computation can be rewritten in this form:

$$\texttt{pure f} \otimes \texttt{c}_1 \otimes \texttt{c}_2 \ldots \otimes \texttt{c}_n$$

Even more explicitly (η-expanding `f`), we might write:

$$\texttt{pure (fun x}_1\ \texttt{x}_2 \ldots \texttt{x}_n \rightarrow \texttt{e)} \otimes \texttt{c}_1 \otimes \texttt{c}_2 \ldots \otimes \texttt{c}_n$$

which corresponds to a `let ... and` expression in OCaml:

```
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

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

Applicative normalisation via the laws

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

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
```

Applicatives built from monads satisfy the laws

The applicative laws follow from the monad laws. Let's prove

```
pure f  ⊗ pure v  ≡  pure (f v)
```

```
pure f ⊗ pure v
```

Applicatives built from monads satisfy the laws

The applicative laws follow from the monad laws. Let's prove

$$\texttt{pure f} \otimes \texttt{pure v} \equiv \texttt{pure (f v)}$$

$\texttt{pure f} \otimes \texttt{pure v}$

$\equiv$ (definition of pure, $\otimes$)

$\texttt{return f} \gg\!= \textbf{fun}\ \texttt{g} \rightarrow \texttt{return v} \gg\!= \textbf{fun}\ \texttt{q} \rightarrow \texttt{return (g q)}$

Applicatives built from monads satisfy the laws

The applicative laws follow from the monad laws. Let's prove

```
pure f ⊗ pure v ≡ pure (f v)
```

```
   pure f ⊗ pure v
≡    (definition of pure, ⊗)
  return f >>= fun g→ return v >>= fun q→ return (g q)
≡    (left identity law (×2))
  (fun g→ (fun q→ return (g q)) v) f
```

Applicatives built from monads satisfy the laws

The applicative laws follow from the monad laws. Let's prove

```
pure f  ⊗ pure v  ≡  pure (f v)
```

```
  pure f ⊗ pure v
≡   (definition of pure, ⊗)
  return f >>= fun g→ return v >>= fun q→ return (g q)
≡   (left identity law (×2))
  (fun g→ (fun q→ return (g q)) v) f
≡   (β reduction (×2))
  return (f v)
```

Applicatives built from monads satisfy the laws

The applicative laws follow from the monad laws. Let's prove

```
pure f ⊗ pure v ≡ pure (f v)
```

```
  pure f ⊗ pure v
≡   (definition of pure, ⊗)
  return f >>= fun g→ return v >>= fun q→ return (g q)
≡   (left identity law (×2))
  (fun g→ (fun q→ return (g q)) v) f
≡   (β reduction (×2))
  return (f v)
≡   (definition of pure)
  pure (f v)
```

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

`pure f` $\otimes$ `pure x`

$\equiv$ (definition of $\otimes$ and `pure`)

`F.pure` $(\otimes_G) \otimes_F$ `F.pure (G.pure f)` $\otimes_F$ `F.pure (G.pure x)`

Composed applicatives are law-abiding

```
pure f ⊗ pure x
```

≡ (definition of ⊗ and `pure`)

`F.pure` $(\otimes_G)\otimes_F$ `F.pure (G.pure f)` $\otimes_F$ `F.pure (G.pure x)`

≡ (homomorphism law for `F` $(\times 2)$)

`F.pure (G.pure f` $\otimes_G$ `G.pure x)`

Composed applicatives are law-abiding

```
 pure f ⊗ pure x
```

≡ (definition of ⊗ and `pure`)

$\texttt{F.pure} \ (\otimes_G\ ) \otimes_F \texttt{F.pure (G.pure f)} \otimes_F \texttt{F.pure (G.pure x)}$

≡ (homomorphism law for `F` ($\times 2$))

$\texttt{F.pure (G.pure f} \otimes_G \texttt{G.pure x)}$

≡ (homomorphism law for `G`)

```
F.pure (G.pure (f x))
```

Composed applicatives are law-abiding

```
 pure f ⊗ pure x
≡  (definition of ⊗ and pure)
 F.pure (⊗_G )⊗_F F.pure (G.pure f)⊗_F F.pure (G.pure x)
≡  (homomorphism law for F (×2))
 F.pure (G.pure f⊗_G G.pure x)
≡  (homomorphism law for G)
 F.pure (G.pure (f x))
≡  (definition of pure)
 pure (f x)
```

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 can't 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
```

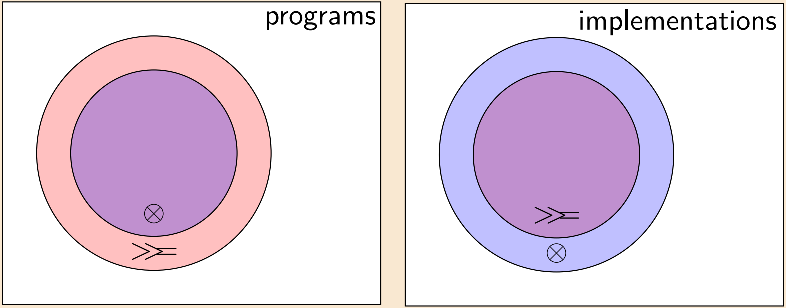
The phantom monoid applicative

```
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
```

```
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.)

monads

let x_1 = e_1 in
let x_2 = e_2 in
...
let x_n = e_n in
  e

applicatives

let x_1 = e_1
and x_2 = e_2
...
and x_n = e_n in
  e