MONADS, COMONADS, AND ALGEBRAIC EFFECTS

HARRISON GRODIN

FEB. 24, 2023

MONADS

MONADS

are type constructors that *work well as outputs*.

MONADS

are representations of *effects*.

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

```
signature MONAD =
  sig
    type 'a m

    val return : 'a -> 'a m
    val bind : 'a m * ('a -> 'b m) -> 'b m

    (* Invariants:
     * bind (return a, f) = f a
     * bind (e, return) = e
     * bind (bind (e, f), g) = bind (e, fn a => bind (f a,
     *)
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a  (* success *)
      | NONE        (* failure *)

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a   (* if e succeeds, run f on the resu
      | NONE   => NONE  (* otherwise, fail
  end
```

EXAMPLE: ERRORS

```
fun divide (x : int, y : int) : int m =
  case y of
    0 => NONE
  | _ => SOME (Int.div (x, y))

val _ : string m =
  bind (divide (10, 3), fn x =>
  bind (Int.fromString "0", fn y =>
  bind (divide (x, y), fn z =>
  return (Int.toString (x + y + z)))))
```

EXAMPLE: ERRORS

```
fun divide (x : int, y : int) : int m =
  case y of
    0 => NONE
  | _ => SOME (Int.div (x, y))

val _ : string m =
  bind (divide (10, 3), fn x =>
  bind (Int.fromString "0", fn y =>
  bind (divide (x, y), fn z =>
  return (Int.toString (x + y + z)))))
```

EXAMPLE: ERRORS

```
fun divide (x : int, y : int) : int m =
  case y of
    0 => NONE
  | _ => SOME (Int.div (x, y))

val _ : string m =
  bind (divide (10, 3), fn x =>
  bind (Int.fromString "0", fn y =>
  bind (divide (x, y), fn z =>
  return (Int.toString (x + y + z)))))
```

EXAMPLE: ERRORS

```
fun divide (x : int, y : int) : int m =
  case y of
    0 => NONE
  | _ => SOME (Int.div (x, y))

val _ : string m =
  bind (divide (10, 3), fn x =>
  bind (Int.fromString "0", fn y =>
  bind (divide (x, y), fn z =>
  return (Int.toString (x + y + z)))))
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e      (* run e  , which prints s1 *)
        val (s2, b) = f a    (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)         (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e      (* run e  , which prints s1 *)
        val (s2, b) = f a    (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)         (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e     (* run e  , which prints s1 *)
        val (s2, b) = f a   (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)        (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e     (* run e  , which prints s1 *)
        val (s2, b) = f a   (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)        (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e      (* run e  , which prints s1 *)
        val (s2, b) = f a    (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)         (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e      (* run e  , which prints s1 *)
        val (s2, b) = f a    (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)         (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e      (* run e  , which prints s1 *)
        val (s2, b) = f a    (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)         (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
structure WriterMonad : MONAD =
  struct
    type 'a m = string * 'a

    fun return (a : 'a) : 'a m = ("", a)

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      let
        val (s1, a) = e    (* run e  , which prints s1 *)
        val (s2, b) = f a  (* run f a, which prints s2 *)
      in
        (s1 ^ s2, b)       (* concatenate printed strings *
      end
  end
```

EXAMPLE: PRINTING

```
fun print (s : string) : unit m = (s, ())

fun add (x : int, y : int) : int m =
  ("adding", x + y)

val _ : int m =
  bind (print "hi", fn () =>
  bind (add (20, 22), fn n =>
  ("done", n)))
```

EXAMPLE: PRINTING

```
fun print (s : string) : unit m = (s, ())

fun add (x : int, y : int) : int m =
  ("adding", x + y)

val _ : int m =
  bind (print "hi", fn () =>
  bind (add (20, 22), fn n =>
  ("done", n)))
```

EXAMPLE: PRINTING

```
fun print (s : string) : unit m = (s, ())

fun add (x : int, y : int) : int m =
  ("adding", x + y)

val _ : int m =
  bind (print "hi", fn () =>
  bind (add (20, 22), fn n =>
  ("done", n)))
```

EXAMPLE: PRINTING

```
fun print (s : string) : unit m = (s, ())

fun add (x : int, y : int) : int m =
  ("adding", x + y)

val _ : int m =
  bind (print "hi", fn () =>
  bind (add (20, 22), fn n =>
  ("done", n)))
```

EXAMPLE: PRINTING

```
fun print (s : string) : unit m = (s, ())

fun add (x : int, y : int) : int m =
  ("adding", x + y)

val _ : int m =
  bind (print "hi", fn () =>
  bind (add (20, 22), fn n =>
  ("done", n)))
```

EXAMPLE: MUTABLE STATE

```
structure StateMonad : MONAD =
  struct
    type 'a m = int -> ('a * int)

    fun return (a : 'a) : 'a m = (fn i => (a, i))

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      fn i =>  (* take in initial state *)
        let
          val (a, i') = e i  (* using initial state *)
        in
          f a i'  (* using new state *)
        end
  end
```

EXAMPLE: MUTABLE STATE

```
structure StateMonad : MONAD =
  struct
    type 'a m = int -> ('a * int)

    fun return (a : 'a) : 'a m = (fn i => (a, i))

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      fn i =>  (* take in initial state *)
        let
          val (a, i') = e i  (* using initial state *)
        in
          f a i'  (* using new state *)
        end
  end
```

EXAMPLE: MUTABLE STATE

```
structure StateMonad : MONAD =
  struct
    type 'a m = int -> ('a * int)

    fun return (a : 'a) : 'a m = (fn i => (a, i))

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      fn i =>  (* take in initial state *)
        let
          val (a, i') = e i  (* using initial state *)
        in
          f a i'  (* using new state *)
        end
  end
```

EXAMPLE: MUTABLE STATE

```
structure StateMonad : MONAD =
  struct
    type 'a m = int -> ('a * int)

    fun return (a : 'a) : 'a m = (fn i => (a, i))

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      fn i =>  (* take in initial state *)
        let
          val (a, i') = e i  (* using initial state *)
        in
          f a i'  (* using new state *)
        end
  end
```

EXAMPLE: MUTABLE STATE

```
structure StateMonad : MONAD =
  struct
    type 'a m = int -> ('a * int)

    fun return (a : 'a) : 'a m = (fn i => (a, i))

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      fn i =>  (* take in initial state *)
        let
          val (a, i') = e i  (* using initial state *)
        in
          f a i'  (* using new state *)
        end
  end
```

EXAMPLE: MUTABLE STATE

```
val get : int m =
  fn i => (i, i)

fun set (i' : int) : unit m =
  fn i => ((), i')

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val   : string m =
```

EXAMPLE: MUTABLE STATE

```
val get : int m =
  fn i => (i, i)

fun set (i' : int) : unit m =
  fn i => ((), i')

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val   : string m =
```

EXAMPLE: MUTABLE STATE

```
val get : int m =
  fn i => (i, i)

fun set (i' : int) : unit m =
  fn i => ((), i')

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val   : string m =
```

EXAMPLE: MUTABLE STATE

```
val get : int m =
  fn i => (i, i)

fun set (i' : int) : unit m =
  fn i => ((), i')

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val _ : string m =
```

EXAMPLE: MUTABLE STATE

```
fun set (i' : int) : unit m =
  fn i => ((), i')

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val _ : string m =
  bind (set 1, fn () =>
  bind (fact 5, fn () =>
```

EXAMPLE: MUTABLE STATE

```
  fn i => ((), i )

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val _ : string m =
  bind (set 1, fn () =>
  bind (fact 5, fn () =>
  bind (get, fn result =>
  return ("result is: " ^ Int.toString result))))
```

EXAMPLE: MUTABLE STATE

```
val get : int m =
  fn i => (i, i)

fun set (i' : int) : unit m =
  fn i => ((), i')

fun fact (i : int) : unit m =
  case i of
    0 => return ()
  | _ =>
      bind (get, fn state =>
      bind (set (i * state), fn () =>
      fact (i - 1)))

val    : string m =
```

MORE EXAMPLES

```
(* nondeterminism *)
type 'a m = 'a list

(* probabilistic sampling *)
type 'a m = (real * 'a) list

(* first-class continuations *)
type 'a m = ('a -> string) -> string

(* unbounded iteration *)
coinductive type 'a m = 'a + 'a m
```

MORE EXAMPLES

```
(* nondeterminism *)
type 'a m = 'a list

(* probabilistic sampling *)
type 'a m = (real * 'a) list

(* first-class continuations *)
type 'a m = ('a -> string) -> string

(* unbounded iteration *)
coinductive type 'a m = 'a + 'a m
```

MORE EXAMPLES

```
(* nondeterminism *)
type 'a m = 'a list

(* probabilistic sampling *)
type 'a m = (real * 'a) list

(* first-class continuations *)
type 'a m = ('a -> string) -> string

(* unbounded iteration *)
coinductive type 'a m = 'a + 'a m
```

MORE EXAMPLES

```
(* nondeterminism *)
type 'a m = 'a list

(* probabilistic sampling *)
type 'a m = (real * 'a) list

(* first-class continuations *)
type 'a m = ('a -> string) -> string

(* unbounded iteration *)
coinductive type 'a m = 'a + 'a m
```

MORE EXAMPLES

```
(* nondeterminism *)
type 'a m = 'a list

(* probabilistic sampling *)
type 'a m = (real * 'a) list

(* first-class continuations *)
type 'a m = ('a -> string) -> string

(* unbounded iteration *)
coinductive type 'a m = 'a + 'a m
```

MORE EXAMPLES

```
(* nondeterminism *)
type 'a m = 'a list

(* probabilistic sampling *)
type 'a m = (real * 'a) list

(* first-class continuations *)
type 'a m = ('a -> string) -> string

(* unbounded iteration *)
coinductive type 'a m = 'a + 'a m
```

TL;DR

TL;DR

TL;DR

- Monads work well as outputs, `'a -> 'b m`

TL;DR

- Monads work well as outputs, `'a -> 'b m`
- Monads are an *abstraction* that represent effects

TL;DR

- Monads work well as outputs, `'a -> 'b m`
- Monads are an *abstraction* that represent effects
- Operations `return` and `bind` generalize imperative programming

ALGEBRAIC EFFECTS

ALGEBRAIC EFFECTS

Effects specified via an *algebraic signature* consisting of:

- a set of *effect operations*, op $\sim A$
- a set of *equations* about sequences of effects

ALGEBRAIC EFFECTS

EXAMPLE: ERRORS

$$\textbf{error} \sim \mathsf{void}$$

ALGEBRAIC EFFECTS

EXAMPLE: PRINTING

$$\mathbf{print}[s] \sim \mathsf{unit}$$

$$\mathbf{print}[""]; e \equiv e$$

$$\mathbf{print}[s_1]; \mathbf{print}[s_2]; e \equiv \mathbf{print}[s_1 \wedge s_2]; e$$

ALGEBRAIC EFFECTS

EXAMPLE: MUTABLE STATE

$$\mathbf{get} \sim \mathsf{int}$$

$$\mathbf{set}[i] \sim \mathsf{unit}$$

$$\mathbf{set}[i_1]; \mathbf{set}[i_2]; e \equiv \mathbf{set}[i_2]; e$$

$$\mathbf{set}[i]; x = \mathbf{get}; e(x) \equiv \mathbf{set}[i]; e(i)$$

$$x = \mathbf{get}; y = \mathbf{get}; e(x, y) \equiv x = \mathbf{get}; e(x, x)$$

ALGEBRAIC EFFECTS VIA MONADS

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

$$\mathbf{op}_1 \sim A_1$$

$$\mathbf{op}_2 \sim A_2$$

$$\mathbf{op}_3 \sim A_3$$

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

```
structure AlgEffMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Op1 of a1 -> 'a m
      | Op2 of a2 -> 'a m
      | Op3 of a3 -> 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a => f a
      | Op1 k    => Op1 (fn y => bind (k y, f))
      | Op2 k    => Op2 (fn y => bind (k y, f))
```

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

```
struct
  datatype 'a m
    = Return of 'a
    | Op1 of a1 -> 'a m
    | Op2 of a2 -> 'a m
    | Op3 of a3 -> 'a m

  fun return (a : 'a) : 'a m = Return a

  fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
    case e of
      Return a => f a
    | Op1 k    => Op1 (fn y => bind (k y, f))
    | Op2 k    => Op2 (fn y => bind (k y, f))
    | Op3 k    => Op3 (fn y => bind (k y, f))
```

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

```
datatype 'a m
  = Return of 'a
  | Op1 of a1 -> 'a m
  | Op2 of a2 -> 'a m
  | Op3 of a3 -> 'a m

fun return (a : 'a) : 'a m = Return a

fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
  case e of
    Return a => f a
  | Op1 k    => Op1 (fn y => bind (k y, f))
  | Op2 k    => Op2 (fn y => bind (k y, f))
  | Op3 k    => Op3 (fn y => bind (k y, f))
end
```

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

```
  datatype 'a m
    = Return of 'a
    | Op1 of a1 -> 'a m
    | Op2 of a2 -> 'a m
    | Op3 of a3 -> 'a m

  fun return (a : 'a) : 'a m = Return a

  fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
    case e of
      Return a => f a
    | Op1 k     => Op1 (fn y => bind (k y, f))
    | Op2 k     => Op2 (fn y => bind (k y, f))
    | Op3 k     => Op3 (fn y => bind (k y, f))
end
```

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

```
datatype 'a m
  = Return of 'a
  | Op1 of a1 -> 'a m
  | Op2 of a2 -> 'a m
  | Op3 of a3 -> 'a m

fun return (a : 'a) : 'a m = Return a

fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
  case e of
    Return a => f a
  | Op1 k    => Op1 (fn y => bind (k y, f))
  | Op2 k    => Op2 (fn y => bind (k y, f))
  | Op3 k    => Op3 (fn y => bind (k y, f))
end
```

ALGEBRAIC EFFECTS VIA MONADS

Every algebraic signature induces a "free" monad.

```
structure AlgEffMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Op1 of a1 -> 'a m
      | Op2 of a2 -> 'a m
      | Op3 of a3 -> 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a => f a
      | Op1 k    => Op1 (fn y => bind (k y, f))
      | Op2 k    => Op2 (fn y => bind (k y, f))
```

error ~ void

error ~ void

```
structure FreeOptionMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Error of void -> 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a => f a
      | Error k  => Error (fn y => bind (k y, f))
  end
```

error ~ void

```
structure FreeOptionMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Error of void -> 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a => f a
      | Error k  => Error (fn y => bind (k y, f))
  end
```

error ~ void

```
structure FreeOptionMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Error of void -> 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a => f a
      | Error k  => Error (fn y => bind (k y, f))
  end
```

error ~ void

```
structure FreeOptionMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Error

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a => f a
      | Error    => Error
  end
```

error ~ void

```
structure OptionMonad : MONAD =
  struct
    datatype 'a m
      = SOME of 'a
      | NONE

    fun return (a : 'a) : 'a m = SOME a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        SOME a => f a
      | NONE   => NONE
  end
```

$$\mathbf{print}[s] \sim \text{unit}$$

print[*s*] ~ unit

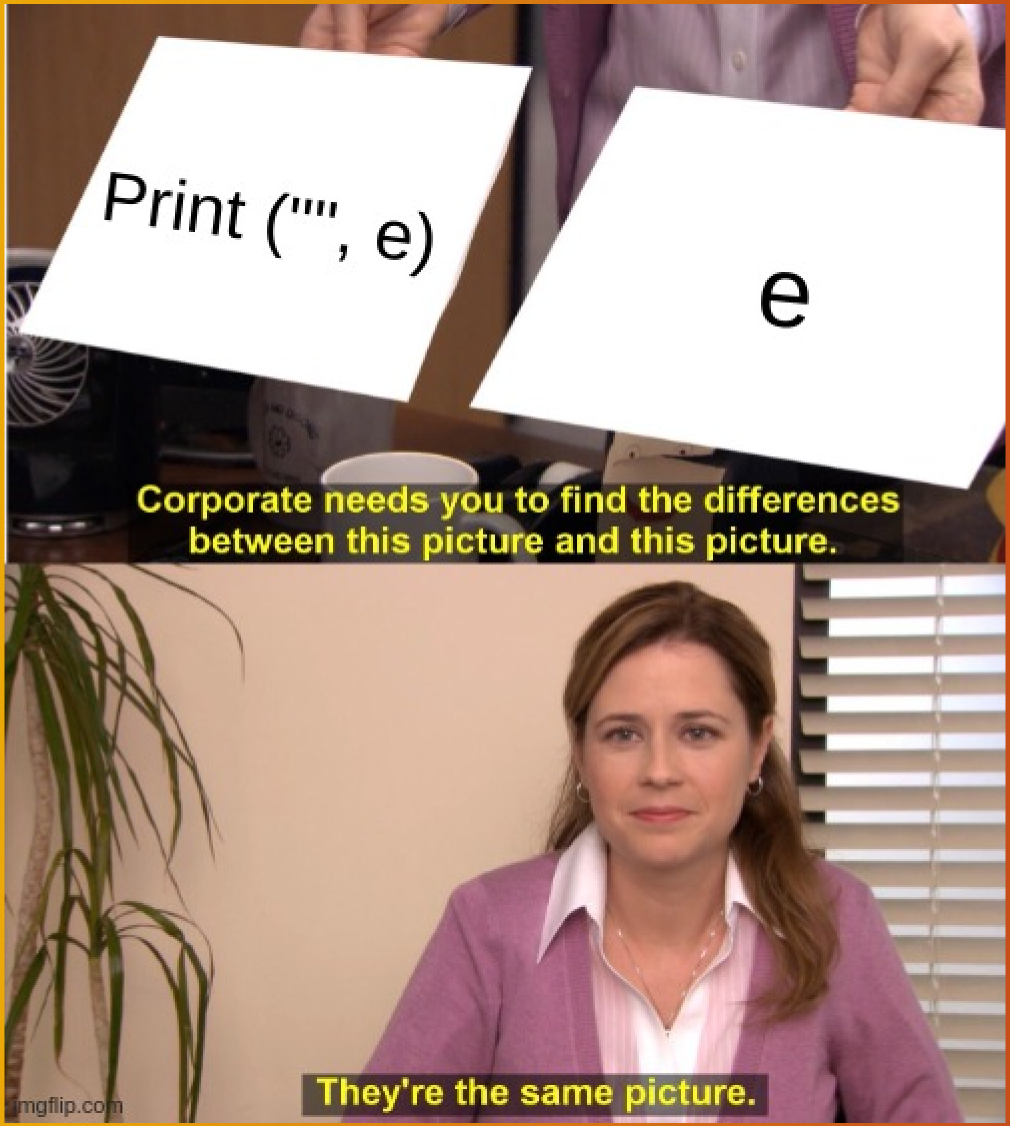
```
structure FreeWriterMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Print of string * 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a      => f a
      | Print (s, k) => Print (s, bind (k, f))
  end
```

print[*s*] ~ unit

```
structure FreeWriterMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Print of string * 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a     => f a
      | Print (s, k) => Print (s, bind (k, f))
  end
```

print[s] ~ unit

```
structure FreeWriterMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Print of string * 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a      => f a
      | Print (s, k) => Print (s, bind (k, f))
  end
```


Print ("", e)
e
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.
imgflip.com

$$\textbf{get} \sim \mathsf{int}$$

$$\textbf{set}[i] \sim \mathsf{unit}$$

$$\mathbf{get} \sim \mathrm{int}$$

$$\mathbf{set}[i] \sim \mathrm{unit}$$

```
structure FreeStateMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Get of int -> 'a m
      | Set of int * 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a    => f a
      | Get k       => Get (fn y => bind (k y, f))
      | Set (i, k) => Set (i, bind (k, f))
  end
```

$$\mathbf{get} \sim \mathrm{int}$$

$$\mathbf{set}[i] \sim \mathrm{unit}$$

```
structure FreeStateMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Get of int -> 'a m
      | Set of int * 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a    => f a
      | Get k       => Get (fn y => bind (k y, f))
      | Set (i, k) => Set (i, bind (k, f))
  end
```

$$\textbf{get} \sim \text{int}$$

$$\textbf{set}[i] \sim \text{unit}$$

```
structure FreeStateMonad : MONAD =
  struct
    datatype 'a m
      = Return of 'a
      | Get of int -> 'a m
      | Set of int * 'a m

    fun return (a : 'a) : 'a m = Return a

    fun bind (e : 'a m, f : 'a -> 'b m) : 'b m =
      case e of
        Return a   => f a
      | Get k      => Get (fn y => bind (k y, f))
      | Set (i, k) => Set (i, bind (k, f))
  end
```

ALGEBRAIC EFFECT HANDLERS

ALGEBRAIC EFFECT HANDLERS

```
datatype 'a m1
  = Return1 of 'a
  | Read1 of string -> 'a m1
  | Print1 of string * 'a m1

datatype 'a m2
  = Return2 of 'a
  | Print2 of string * 'a m2

fun handler (e : int m1) : int m2 =
  case e of
    Return1 x      => Print2 ("😈", Return2 (x + 1))
  | Read1 k        => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)
```

ALGEBRAIC EFFECT HANDLERS

```
datatype 'a m1
  = Return1 of 'a
  | Read1 of string -> 'a m1
  | Print1 of string * 'a m1

datatype 'a m2
  = Return2 of 'a
  | Print2 of string * 'a m2

fun handler (e : int m1) : int m2 =
  case e of
    Return1 x      => Print2 ("😈", Return2 (x + 1))
  | Read1 k        => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)
```

ALGEBRAIC EFFECT HANDLERS

```
datatype 'a m1
  = Return1 of 'a
  | Read1 of string -> 'a m1
  | Print1 of string * 'a m1

datatype 'a m2
  = Return2 of 'a
  | Print2 of string * 'a m2

fun handler (e : int m1) : int m2 =
  case e of
    Return1 x      => Print2 ("😈", Return2 (x + 1))
  | Read1 k        => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)
```

ALGEBRAIC EFFECT HANDLERS

```

datatype 'a m2
  = Return2 of 'a
  | Print2 of string * 'a m2

fun handler (e : int m1) : int m2 =
  case e of
    Return1 x      => Print2 ("😈", Return2 (x + 1))
  | Read1 k        => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)

val example : int m1 =
  Print1 ("live",
  Read1 (fn s =>
  Print1 (s,
```

ALGEBRAIC EFFECT HANDLERS

```
  case e of
    Return1 x      => Print2 ("😈", Return2 (x + 1))
  | Read1 k        => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)

val example : int m1 =
  Print1 ("live",
  Read1 (fn s =>
  Print1 (s,
  Return1 41)))

val example' : int m2 =
  handler example
(* Print2 ("evil",
   Print2 ("spam",
```

ALGEBRAIC EFFECT HANDLERS

```
  | Read1 k      => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)

val example : int m1 =
  Print1 ("live",
  Read1 (fn s =>
  Print1 (s,
  Return1 41)))

val example' : int m2 =
  handler example
(* Print2 ("evil",
   Print2 ("spam",
   Print2 ("😈",
   Return2 42))) *)
```

ALGEBRAIC EFFECT HANDLERS

```
  | Read1 k        => handler (k "maps")
  | Print1 (s, k) => Print2 (String.rev s, handler k)

val example : int m1 =
  Print1 ("live",
  Read1 (fn s =>
  Print1 (s,
  Return1 41)))

val example' : int m2 =
  handler example
(* Print2 ("evil",
   Print2 ("spam",
   Print2 ("😈",
   Return2 42))) *)
```

ALGEBRAIC EFFECT HANDLERS

```
datatype 'a m
  = Return of 'a
  | Error
  | Flip of bool -> 'a m

val example : (bool * bool) m =
  Flip (fn b1 =>
  Flip (fn b2 =>
  if b1 orelse b2
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a     => SOME a
```

ALGEBRAIC EFFECT HANDLERS

```
datatype 'a m
  = Return of 'a
  | Error
  | Flip of bool -> 'a m

val example : (bool * bool) m =
  Flip (fn b1 =>
  Flip (fn b2 =>
  if b1 orelse b2
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a      => SOME a
```

ALGEBRAIC EFFECT HANDLERS

```
datatype 'a m
  = Return of 'a
  | Error
  | Flip of bool -> 'a m

val example : (bool * bool) m =
  Flip (fn b1 =>
  Flip (fn b2 =>
  if b1 orelse b2
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a     => SOME a
  | Error        => NONE
```

ALGEBRAIC EFFECT HANDLERS

```
val example : (bool * bool) m =
  Flip (fn b1 =>
  Flip (fn b2 =>
  if b1 orelse b2
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a      => SOME a
  | Error         => NONE
  | Flip k        =>
      case handlerOne (k false) of
        SOME a => SOME a
      | NONE   => handlerOne (k true)
```

ALGEBRAIC EFFECT HANDLERS

```
  Flip (fn b1 =>
  Flip (fn b2 =>
  if b1 orelse b2
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a      => SOME a
  | Error         => NONE
  | Flip k        =>
      case handlerOne (k false) of
        SOME a => SOME a
      | NONE   => handlerOne (k true)

```

ALGEBRAIC EFFECT HANDLERS

```
  Flip (fn b2 =>
  if b1 orelse b2
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a      => SOME a
  | Error         => NONE
  | Flip k        =>
      case handlerOne (k false) of
        SOME a => SOME a
      | NONE   => handlerOne (k true)

fun handlerAll (e : 'a m) : 'a list =
```

ALGEBRAIC EFFECT HANDLERS

```
    then Return (b1, b2)
    else Error))

fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a      => SOME a
  | Error         => NONE
  | Flip k        =>
      case handlerOne (k false) of
        SOME a => SOME a
      | NONE   => handlerOne (k true)

fun handlerAll (e : 'a m) : 'a list =
  case e of
```

ALGEBRAIC EFFECT HANDLERS

```
fun handlerOne (e : 'a m) : 'a option =
  case e of
    Return a      => SOME a
  | Error         => NONE
  | Flip k        =>
      case handlerOne (k false) of
        SOME a => SOME a
      | NONE   => handlerOne (k true)

fun handlerAll (e : 'a m) : 'a list =
  case e of
    Return a      => [a]
  | Error         => []
  | Flip k        => handlerAll (k false) @ handlerAll (k tr
```

NOT ALL MONADS ARE ALGEBRAIC. ⚠️

(continuations, unbounded iteration, etc.)

TL;DR

TL;DR

TL;DR

- Algebraic effects are easy to *specify* and *compose*

TL;DR

- Algebraic effects are easy to *specify* and *compose*
- An algebraic effect signature induces a "free" monad

TL;DR

- Algebraic effects are easy to *specify* and *compose*
- An algebraic effect signature induces a "free" monad
- Not all monads are algebraic...

TL;DR

- Algebraic effects are easy to *specify* and *compose*
- An algebraic effect signature induces a "free" monad
- Not all monads are algebraic...
- ...but the ones that are support *handlers*

COMONADS

COMONADS

are type constructors that *work well as inputs*.

COMONADS

are representations of *"coeffects"/environments*.

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

```
signature COMONAD =
  sig
    type 'a w

    val extract : 'a w -> 'a
    val extend : 'a w * ('a w -> 'b) -> 'b w

    (* Invariants:
     * extract (extend (e, f)) = f e
     * extend (e, extract) = e
     * extend (extend (e, f), g) = extend (e, fn a => exten
     *)
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
structure EnvironmentComonad : COMONAD =
  struct
    type 'a w =
      { extract     : 'a
      , environment : string
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract     = f e
      , environment = #environment e
      }
  end
```

EXAMPLE: ENVIRONMENT

```
fun getHOME (e : 'a w) : string = #environment e

fun makePath (e : string w) : string =
  getHOME e ^ "/" ^ extract e
```

EXAMPLE: ENVIRONMENT

```
fun getHOME (e : 'a w) : string = #environment e

fun makePath (e : string w) : string =
  getHOME e ^ "/" ^ extract e
```

EXAMPLE: ENVIRONMENT

```
fun getHOME (e : 'a w) : string = #environment e

fun makePath (e : string w) : string =
  getHOME e ^ "/" ^ extract e
```

EXAMPLE: ENVIRONMENT

```
fun getHOME (e : 'a w) : string = #environment e

fun makePath (e : string w) : string =
  getHOME e ^ "/" ^ extract e
```

EXAMPLE: HISTORY STREAM

```
structure TraceComonad : COMONAD =
  struct
    type 'a w = nat -> 'a

    fun extract (e : 'a w) : 'a = e 0

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      fn n1 => f (fn n2 => e (n1 + n2))
  end
```

EXAMPLE: HISTORY STREAM

```
structure TraceComonad : COMONAD =
  struct
    type 'a w = nat -> 'a

    fun extract (e : 'a w) : 'a = e 0

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      fn n1 => f (fn n2 => e (n1 + n2))
  end
```

EXAMPLE: HISTORY STREAM

```
structure TraceComonad : COMONAD =
  struct
    type 'a w = nat -> 'a

    fun extract (e : 'a w) : 'a = e 0

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      fn n1 => f (fn n2 => e (n1 + n2))
  end
```

EXAMPLE: HISTORY STREAM

```
structure TraceComonad : COMONAD =
  struct
    type 'a w = nat -> 'a

    fun extract (e : 'a w) : 'a = e 0

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      fn n1 => f (fn n2 => e (n1 + n2))
  end
```

EXAMPLE: HISTORY STREAM

```
structure TraceComonad : COMONAD =
  struct
    type 'a w = nat -> 'a

    fun extract (e : 'a w) : 'a = e 0

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      fn n1 => f (fn n2 => e (n1 + n2))
  end
```

EXAMPLE: HISTORY STREAM

```
fun prev (e : 'a w) (n : nat) : 'a = e n

fun recentAverage (e : real w) : real =
  (prev e 0 + prev e 1 + prev e 2) / 3.0

fun movingAverage (e : real w) : real w =
  extend (e, recentAverage)
```

EXAMPLE: HISTORY STREAM

```
fun prev (e : 'a w) (n : nat) : 'a = e n

fun recentAverage (e : real w) : real =
  (prev e 0 + prev e 1 + prev e 2) / 3.0

fun movingAverage (e : real w) : real w =
  extend (e, recentAverage)
```

EXAMPLE: HISTORY STREAM

```
fun prev (e : 'a w) (n : nat) : 'a = e n

fun recentAverage (e : real w) : real =
  (prev e 0 + prev e 1 + prev e 2) / 3.0

fun movingAverage (e : real w) : real w =
  extend (e, recentAverage)
```

EXAMPLE: HISTORY STREAM

```
fun prev (e : 'a w) (n : nat) : 'a = e n

fun recentAverage (e : real w) : real =
  (prev e 0 + prev e 1 + prev e 2) / 3.0

fun movingAverage (e : real w) : real w =
  extend (e, recentAverage)
```

EXAMPLE: HISTORY STREAM

```
fun prev (e : 'a w) (n : nat) : 'a = e n

fun recentAverage (e : real w) : real =
  (prev e 0 + prev e 1 + prev e 2) / 3.0

fun movingAverage (e : real w) : real w =
  extend (e, recentAverage)
```

TL;DR

TL;DR

TL;DR

- Comonads work well as inputs, `'a w -> 'b`

TL;DR

- Comonads work well as inputs, `'a w -> 'b`
- Comonads are an *abstraction* that represent environments ("having extra stuff"), with `extract` and `extend`

COMONADS AND ALGEBRAIC EFFECTS

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

$$\mathbf{op}_1 \sim A_1$$

$$\mathbf{op}_2 \sim A_2$$

$$\mathbf{op}_3 \sim A_3$$

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

```
structure AlgEffComonad : COMONAD =
  struct
    coinductive type 'a w =
      { extract : 'a
      , op1 : a1 * 'a w
      , op2 : a2 * 'a w
      , op3 : a3 * 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , op1 = (fst (#op1 e), extend (snd (#op1 e), f))
      , op2 = (fst (#op2 e), extend (snd (#op2 e), f))
```

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

```
coinductive type 'a w =
  { extract : 'a
  , op1 : a1 * 'a w
  , op2 : a2 * 'a w
  , op3 : a3 * 'a w
  }

fun extract (e : 'a w) : 'a = #extract e

fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
  { extract = f e
  , op1 = (fst (#op1 e), extend (snd (#op1 e), f))
  , op2 = (fst (#op2 e), extend (snd (#op2 e), f))
  , op3 = (fst (#op3 e), extend (snd (#op3 e), f))
  }
```

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

```
      { extract : 'a
      , op1 : a1 * 'a w
      , op2 : a2 * 'a w
      , op3 : a3 * 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , op1 = (fst (#op1 e), extend (snd (#op1 e), f))
      , op2 = (fst (#op2 e), extend (snd (#op2 e), f))
      , op3 = (fst (#op3 e), extend (snd (#op3 e), f))
      }
  end
```

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

```
    { extract : 'a
    , op1 : a1 * 'a w
    , op2 : a2 * 'a w
    , op3 : a3 * 'a w
    }

  fun extract (e : 'a w) : 'a = #extract e

  fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
    { extract = f e
    , op1 = (fst (#op1 e), extend (snd (#op1 e), f))
    , op2 = (fst (#op2 e), extend (snd (#op2 e), f))
    , op3 = (fst (#op3 e), extend (snd (#op3 e), f))
    }
end
```

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

```
    { extract : 'a
    , op1 : a1 * 'a w
    , op2 : a2 * 'a w
    , op3 : a3 * 'a w
    }

  fun extract (e : 'a w) : 'a = #extract e

  fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
    { extract = f e
    , op1 = (fst (#op1 e), extend (snd (#op1 e), f))
    , op2 = (fst (#op2 e), extend (snd (#op2 e), f))
    , op3 = (fst (#op3 e), extend (snd (#op3 e), f))
    }
end
```

COMONADS AND ALGEBRAIC EFFECTS

Every algebraic signature induces a "cofree" comonad.

```
structure AlgEffComonad : COMONAD =
  struct
    coinductive type 'a w =
      { extract : 'a
      , op1 : a1 * 'a w
      , op2 : a2 * 'a w
      , op3 : a3 * 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , op1 = (fst (#op1 e), extend (snd (#op1 e), f))
      , op2 = (fst (#op2 e), extend (snd (#op2 e), f))
```

$$\textbf{print}[c] \sim \text{unit}$$

$\mathbf{print}[c] \sim \mathrm{unit}$

```
structure CofreeWriterComonad : COMONAD =
  struct
    coinductive type 'a w =
      { extract : 'a
      , print : char -> 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , print = fn c => extend (#print e c, f)
      }
  end
```

$$\mathbf{print}[c] \sim \text{unit}$$

```
structure CofreeWriterComonad : COMONAD =
  struct
    coinductive type 'a w =
      { extract : 'a
      , print : char -> 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , print = fn c => extend (#print e c, f)
      }
  end
```

$$\mathbf{print}[c] \sim \text{unit}$$

```
structure CofreeWriterComonad : COMONAD =
  struct
    coinductive type 'a w =
      { extract : 'a
      , print : char -> 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , print = fn c => extend (#print e c, f)
      }
  end
```

$\mathbf{print}[c] \sim \text{unit}$

```
coinductive type 'a w =
  { extract : 'a
  , print : char -> 'a w
  }
```

$$W(A) = A \times (\Sigma \to W(A))$$

$$\textbf{print}[c] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , print : char -> 'a w
  }
```

$$W(A) = A \times (\Sigma \to W(A))$$

```
fun parity_q (b : bool) : bool w =
  { extract = b
  , print = fn #"q" => parity_q (not b) | _ => parity_q b
  }

val even_q = parity_q true
```

$$\mathbf{print}[c] \sim \mathrm{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , print : char -> 'a w
  }
```

$$W(A) = A \times (\Sigma \rightarrow W(A))$$

```
fun parity_q (b : bool) : bool w =
  { extract = b
  , print = fn #"q" => parity_q (not b) | _ => parity_q b
  }

val even_q = parity_q true
```

$$\mathbf{print}[c] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , print : char -> 'a w
  }
```

$$W(A) = A \times (\Sigma \rightarrow W(A))$$

```
fun parity_q (b : bool) : bool w =
  { extract = b
  , print = fn #"q" => parity_q (not b) | _ => parity_q b
  }

val even_q = parity_q true
```

$$\textbf{print}[c] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , print : char -> 'a w
  }
```

$$W(A) = A \times (\Sigma \to W(A))$$

```
fun parity_q (b : bool) : bool w =
  { extract = b
  , print = fn #"q" => parity_q (not b) | _ => parity_q b
  }

val even_q = parity_q true
```

This is an *automaton*!

$$\mathbf{get} \sim \mathsf{int}$$

$$\mathbf{set}[i] \sim \mathsf{unit}$$

$$\mathbf{get} \sim \mathrm{int}$$

$$\mathbf{set}[i] \sim \mathrm{unit}$$

```
structure CofreeStateComonad : COMONAD =
  struct
    coinductive type 'a w =
      { extract : 'a
      , get : int * 'a w
      , set : int -> 'a w
      }

    fun extract (e : 'a w) : 'a = #extract e

    fun extend (e : 'a w, f : 'a w -> 'b) : 'b w =
      { extract = f e
      , get = (fst (#get e), extend (snd (#get e), f))
      , set = fn i => extend (#set e i, f)
      }
```

$$\mathbf{get} \sim \mathsf{int}$$

$$\mathbf{set}[i] \sim \mathsf{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

$$\textbf{get} \sim \text{int}$$

$$\textbf{set}[i] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

$$\textbf{get} \sim \text{int}$$

$$\textbf{set}[i] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

$$\textbf{get} \sim \text{int}$$

$$\textbf{set}[i] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

$$\mathbf{get} \sim \mathrm{int}$$

$$\mathbf{set}[i] \sim \mathrm{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

$$\mathbf{get} \sim \text{int}$$

$$\mathbf{set}[i] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

$$\textbf{get} \sim \text{int}$$

$$\textbf{set}[i] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

$$\textbf{get} \sim \text{int}$$

$$\textbf{set}[i] \sim \text{unit}$$

```
coinductive type 'a w =
  { extract : 'a
  , get : int * 'a w
  , set : int -> 'a w
  }
```

```
fun machine (state : int) : string w =
  { extract = "final: " ^ Int.toString state
  , get = (state, machine state)
  , set = fn state' => machine state'
  }

val example = machine 42
```

TL;DR

TL;DR

TL;DR

- An algebraic effect signature induces a "cofree" comonad

TL;DR

- An algebraic effect signature induces a "cofree" comonad
- The elements of this comonad behave like machines

FREE MONADS AND COFREE COMONADS FOR ALGEBRAIC EFFECTS

(Software 🤝 Hardware)

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a              =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a              =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a             =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a             =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a              =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a               =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
val software : unit m =
  Print (#"q", Print (#"r", Print (#"q", Return ())))

val hardware : bool w = even_q

val ((), true) = execute (software, hardware)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a              =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
val software : unit m =
  Print (#"q", Print (#"r", Print (#"q", Return ())))

val hardware : bool w = even_q

val ((), true) = execute (software, hardware)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a              =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
val software : unit m =
  Print (#"q", Print (#"r", Print (#"q", Return ())))

val hardware : bool w = even_q

val ((), true) = execute (software, hardware)
```

```
datatype 'a m
  = Return of 'a
  | Print of char * 'a m

coinductive type 'b w =
  { extract : 'b
  , print : char -> 'b w
  }
```

```
fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =
  case software of
    Return a                =>
      (a, extract hardware)
  | Print (c, software') =>
      execute (software', #print hardware c)
```

```
val software : unit m =
  Print (#"q", Print (#"r", Print (#"q", Return ())))

val hardware : bool w = even_q

val ((), true) = execute (software, hardware)
```

USE CASE: DATA STRUCTURES

$$\mathbf{add}[k, v] \sim \text{unit}$$
$$\mathbf{remove}[k] \sim \text{unit}$$
$$\mathbf{find}[k] \sim V + 1$$

USE CASE: DATA STRUCTURES

$$\mathbf{add}[k, v] \sim \text{unit}$$

$$\mathbf{remove}[k] \sim \text{unit}$$

$$\mathbf{find}[k] \sim V + 1$$

```
datatype 'a m
  = Return of 'a
  | Add of key * value * 'a m
  | Remove of key * 'a m
  | Find of key * (value option -> 'a m)

coinductive type 'b w =
  { extract : 'b
  , add : key * value -> 'b w
  , remove : key -> 'b w
  , find : key -> (value option * 'b w)
  }
```

USE CASE: DATA STRUCTURES

$$\mathbf{add}[k, v] \sim \text{unit}$$

$$\mathbf{remove}[k] \sim \text{unit}$$

$$\mathbf{find}[k] \sim V + 1$$

```
datatype 'a m
  = Return of 'a
  | Add of key * value * 'a m
  | Remove of key * 'a m
  | Find of key * (value option -> 'a m)

coinductive type 'b w =
  { extract : 'b
  , add : key * value -> 'b w
  , remove : key -> 'b w
  , find : key -> (value option * 'b w)
  }
```

USE CASE: DATA STRUCTURES

$$\textbf{add}[k, v] \sim \text{unit}$$

$$\textbf{remove}[k] \sim \text{unit}$$

$$\textbf{find}[k] \sim V + 1$$

```
datatype 'a m
  = Return of 'a
  | Add of key * value * 'a m
  | Remove of key * 'a m
  | Find of key * (value option -> 'a m)

coinductive type 'b w =
  { extract : 'b
  , add : key * value -> 'b w
  , remove : key -> 'b w
  , find : key -> (value option * 'b w)
  }
```

USE CASE: DATA STRUCTURES

```
type key   = int
and  value = string

val software : bool m =
  Add (3, "cow",
  Find (8, fn opt =>
  case opt of
    NONE   => Return false
  | SOME k => Remove (8, Return true)))

(* using red-black trees *)
fun hardwareRBT (t : RBT.t) : int w =
  { extract = RBT.size t
  , add = fn (k, v) => hardwareRBT (RBT.insert t (k, v))
  , remove = fn k => hardwareRBT (RBT.remove t k)
```

USE CASE: DATA STRUCTURES

```
    NONE   => Return false
  | SOME k => Remove (8, Return true)))

(* using red-black trees *)
fun hardwareRBT (t : RBT.t) : int w =
  { extract = RBT.size t
  , add = fn (k, v) => hardwareRBT (RBT.insert t (k, v))
  , remove = fn k => hardwareRBT (RBT.remove t k)
  , find = fn k => (RBT.find t k, hardwareRBT t)
  }
val hardware = hardwareRBT RBT.empty

fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =

val (false, 1) = execute (software, hardware)
```

USE CASE: DATA STRUCTURES

```
      NONE   => Return false
    | SOME k => Remove (8, Return true)))

(* using red-black trees *)
fun hardwareRBT (t : RBT.t) : int w =
  { extract = RBT.size t
  , add = fn (k, v) => hardwareRBT (RBT.insert t (k, v))
  , remove = fn k => hardwareRBT (RBT.remove t k)
  , find = fn k => (RBT.find t k, hardwareRBT t)
  }
val hardware = hardwareRBT RBT.empty

fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =

val (false, 1) = execute (software, hardware)
```

USE CASE: DATA STRUCTURES

```
      NONE    => Return false
    | SOME k => Remove (8, Return true)))

(* using red-black trees *)
fun hardwareRBT (t : RBT.t) : int w =
  { extract = RBT.size t
  , add = fn (k, v) => hardwareRBT (RBT.insert t (k, v))
  , remove = fn k => hardwareRBT (RBT.remove t k)
  , find = fn k => (RBT.find t k, hardwareRBT t)
  }
val hardware = hardwareRBT RBT.empty

fun execute (software : 'a m, hardware : 'b w) : 'a * 'b =

val (false, 1) = execute (software, hardware)
```

USE CASE: DATA STRUCTURES

```
type key   = int
and  value = string

val software : bool m =
  Add (3, "cow",
  Find (8, fn opt =>
  case opt of
    NONE   => Return false
  | SOME k => Remove (8, Return true)))

(* using red-black trees *)
fun hardwareRBT (t : RBT.t) : int w =
  { extract = RBT.size t
  , add = fn (k, v) => hardwareRBT (RBT.insert t (k, v))
  , remove = fn k => hardwareRBT (RBT.remove t k)
```

CONCLUSION

CONCLUSION

CONCLUSION

- Monads represent outputs/effects/instructions

CONCLUSION

- Monads represent outputs/effects/instructions
- Comonads represent inputs/environments/machines

CONCLUSION

- Monads represent outputs/effects/instructions
- Comonads represent inputs/environments/machines
- Algebraic effects make it easy to specify the common cases of free monads and cofree comonads (and are handleable!)

CONCLUSION

- Monads represent outputs/effects/instructions
- Comonads represent inputs/environments/machines
- Algebraic effects make it easy to specify the common cases of free monads and cofree comonads (and are handleable!)

You can 🧑‍💻 *code* with these abstractions and/or use them for 🧑‍🔬 *semantics*!

REFERENCES

Bauer, Andrej. 2018. "What Is Algebraic About Algebraic Effects and Handlers?"

Katsumata, Shin-ya, Exequiel Rivas, and Tarmo Uustalu. 2020. "Interaction Laws of Monads and Comonads."

Pretnar, Matija. 2015. "An Introduction to Algebraic Effects and Handlers."