Last time: generic programming

\[ \text{val } (=) : \{D: DATA\} \rightarrow D.t \rightarrow D.t \rightarrow \text{bool} \]
This time: monads etc., continued

\[ \Rightarrow \quad \otimes \quad \text{effect E} \]
Recap: monads, bind and let

An imperative program

```ocaml
let id = !counter in
let () = counter := id + 1 in
  string_of_int id
```

A monadic program

```ocaml
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
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:

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 $\gg$: \[ \forall a, b. \ (a ightarrow b) ightarrow b \]

$\forall a \rightarrow b. t$: a function that builds a computation

(Against) the type of $\diamond$: \[ \forall a, b. \ (a ightarrow b) ightarrow b \]

$(a ightarrow b) \ t$: a computation that builds a function

The actual type of $\diamond$: \[ (a ightarrow b) \ t \rightarrow a \ t \rightarrow b \]
Applicative normal forms

\[ \text{pure } f \otimes c_1 \otimes c_2 \ldots \otimes c_n \]

\[ \text{pure } (\text{fun } x_1 \ x_2 \ldots x_n \rightarrow e) \otimes c_1 \otimes c_2 \ldots \otimes c_n \]

\[
\begin{align*}
\text{let } x_1 &= c_1 \\
\text{and } x_2 &= c_2 \\
\ldots \\
\text{and } x_n &= c_n \\
\text{in } e
\end{align*}
\]
Applicative normalisation via the laws

\[ \text{pure } f \otimes (\text{pure } g \otimes \text{fresh}_\text{name}) \otimes \text{fresh}_\text{name} \]
Applicative normalisation via the laws

\[
pure \, f \odot (pure \, g \odot fresh\_name) \odot fresh\_name \\
\equiv \quad (\text{composition law}) \\
(pure \, \text{compose} \odot pure \, f \odot pure \, g \odot fresh\_name) \odot fresh\_name
\]
Applicative normalisation via the laws

\[
pure \, f \otimes (pure \, g \otimes fresh\_name) \otimes fresh\_name \\
\equiv \quad \text{(composition law)} \\
(pure \, \text{compose} \otimes pure \, f \otimes pure \, g \otimes fresh\_name) \otimes fresh\_name \\
\equiv \quad \text{(homomorphism law (×2))} \\
pure \, (\text{compose} \, f \, g) \otimes fresh\_name \otimes fresh\_name
\]
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

```ocaml
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 ⊗ pure x
Composed applicatives are law-abiding

\[ \text{pure } f \otimes \text{pure } x \]
\[ \equiv \text{ (definition of } \otimes \text{ and pure)} \]
\[ F.\text{pure } (\otimes_G ) \otimes_F F.\text{pure } (G.\text{pure } f) \otimes_F F.\text{pure } (G.\text{pure } x) \]
Composed applicatives are law-abiding

\[
pure f \otimes pure x
\]  
\[\equiv \quad (\text{definition of } \otimes \text{ and pure})\]

\[
F.pure (\otimes_G ) \otimes_F F.pure (G.pure f) \otimes_F F.pure (G.pure x)
\]

\[\equiv \quad (\text{homomorphism law for } F (\times 2))\]

\[
F.pure (G.pure f \otimes_G G.pure x)
\]
Composed applicatives are law-abiding

\[
\begin{align*}
\text{pure } f & \otimes \text{pure } x \\
\equiv & \quad \text{(definition of } \otimes \text{ and pure)} \\
F.\text{pure } (\otimes_G ) & \otimes_F F.\text{pure } (G.\text{pure } f) \otimes_F F.\text{pure } (G.\text{pure } x) \\
\equiv & \quad \text{(homomorphism law for } F \text{ (×2))} \\
F.\text{pure } (G.\text{pure } f \otimes_G G.\text{pure } x) & \\
\equiv & \quad \text{(homomorphism law for } G) \\
F.\text{pure } (G.\text{pure } (f \times))
\end{align*}
\]
Composed applicatives are law-abiding

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

```ocaml
defun fresh_name : string IState.t =
  get >>> fun i ➞
  put (i + 1) >>> fun () ➞
  return (Printf.printf "x%d" i)
```

**Solution:** introduce `fresh_name` as a primitive effect:

```ocaml
implicit module NameA : sig
  module Applicative : APPLICATIVE
  val fresh_name : string Applicative.t
end = ...
```
Traversing with namer

```ocaml
defunction
    label_tree (a tree → string tree NameA.t)
    let rec label_tree : 'a tree → string tree NameA.t =
        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

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

\[
\text{let } x_1 = e_1 \text{ in } \\
\text{let } x_2 = e_2 \text{ in } \\
\text{...} \\
\text{let } x_n = e_n \text{ in } \\
\]

\[
\text{e}
\]

applicatives

\[
\text{let } x_1 = e_1 \\
\text{and } x_2 = e_2 \\
\text{...} \\
\text{and } x_n = e_n \text{ in } \\
\]

\[
\text{e}
\]
Algebraic effects and handlers

(effect E)
Extending \texttt{match} for exceptions

Possible outcomes of \texttt{match}

\begin{verbatim}
match f v with
| A x \rightarrow g x
| B y \rightarrow h y
| ...
\end{verbatim}
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*

```latex
match f v with
  | A x \rightarrow g x
  | B y \rightarrow h y
  | ...
```

- **f v evaluates to** A x: *evaluate* g x
- **f v evaluates to** B y: *evaluate* h y

...
Extending \texttt{match} for exceptions

Possible outcomes of \texttt{match}

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

\textit{f v} evaluates to \texttt{A x}: \hspace{1em} \texttt{evaluate g x}

\textit{f v} evaluates to \texttt{B y}: \hspace{1em} \texttt{evaluate h y}

... 

\textit{f v} raises an \texttt{exception} \texttt{E}: \hspace{1em} \texttt{raise E}
Extending \texttt{match} for exceptions

Write:

\begin{verbatim}
match f v with
  | A x -> g x
  | B y -> h y
  | ... 
  | exception (E z) -> j z
\end{verbatim}

E.g. search an association list \texttt{l} (type \texttt{(string * bool) list}):

\begin{verbatim}
match List.assoc s l with
  | true -> "found (True)"
  | false -> "found (False)"
  | exception Not_found -> "not found"
\end{verbatim}
Extending \texttt{match} for effects

Possible outcomes of \texttt{match}

\begin{verbatim}
match f v with
| A x \rightarrow g x
| B y \rightarrow h y
| exception (E z) \rightarrow j z
| ...
\end{verbatim}

\texttt{f v evaluates to A x: evaluate g x}

\texttt{f v evaluates to B y: evaluate h y}

\texttt{f v raises an exception E: raise E}

\texttt{...}
Extending `match` for effects

Possible outcomes of `match`

```latex
match f v with
| A x \rightarrow g x
| B y \rightarrow h y
| \text{exception} (E z) \rightarrow 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

\[ \text{exception } E: s \rightarrow \text{exn} \quad (\text{means} \quad \text{type } \text{exn} += E: s \rightarrow \text{exn}) \]

Raising exceptions

\[ \text{val } \text{raise} : \text{exn} \rightarrow 'b \]

Handling exceptions

\[ \text{match } e \text{ with} \]
\[ \ldots \]
\[ \mid \text{exception } (E \ x) \rightarrow \ldots \]
Elements of effects

Effects

\[
\text{effect } E : s \rightarrow t \quad (\text{means type } \_ \text{ eff } += E : s \rightarrow t \text{ eff})
\]

Performing effects

\[
\text{val perform : } 'a \text{ eff } \rightarrow 'a
\]

Handling effects

\[
\text{match e with}
\]
\[
\text{...}
\]
\[
\mid \text{effect } (E \ x) \ k \rightarrow ...
\]

Running continuations

\[
\text{val continue : } ('a, 'b) \text{ continuation } \rightarrow 'a \rightarrow 'b
\]
Using effects: yet another ocaml fork

modular implicits

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

effects

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

staging (next week)

```bash
opam switch 4.02.1+modular-implicits-ber
```
Example: exceptions as an effect

Define the effect and a function to perform the effect:

\[
\text{effect}\ \text{Raise} : \text{exn} \to 'a \\
\text{let}\ raise\ e = \text{perform}(\text{Raise} e)
\]

Define a function to handle the effect:

\[
\text{let}_\text{try}\ f\ \text{handle} = \\
\text{match} f()\ \text{with} \\
| v \to v \\
| \text{effect} (\text{Raise} e)\ k \to (*\ \text{discard}\ k!\ *)\ \text{handle}\ e
\]

Program in direct (non-monadic) style:

\[
\text{let}_\text{rec}\ \text{assoc}\ x = \text{function} \\
| [] \to \text{raise}\ \text{Not\_found} \\
| (k,v)::t \to \text{if}\ k = x\ \text{then}\ v\ \text{else}\ \text{assoc}\ x\ t
\]

\[
_\text{try}\ (\text{fun}() \to \text{Some}(\text{assoc}\ 3\ 1)) \\
(\text{fun}\ ex \to \text{None})
\]
Recap: state as a monad

The type of computations:

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

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

```ocaml
let return v s = (s, v)
let (>>=) m k s = let s’, a = m s in k a s'
```

Signatures of primitive effects:

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

Primitive effects and a `run` function:

```ocaml
let get s = (s, s)
let put s’ _ = (s’, ())
let runState m init = m init
```
Example: state as an effect

Primitive effects:

\[
\begin{align*}
\text{effect} & \quad \text{Put} : \text{state} \rightarrow \text{unit} \\
\text{effect} & \quad \text{Get} : \text{state}
\end{align*}
\]

Functions to perform effects:

\[
\begin{align*}
\text{let} & \quad \text{put} \ v = \text{perform} \ (\text{Put} \ v) \\
\text{let} & \quad \text{get} \ () = \text{perform} \ \text{Get}
\end{align*}
\]

A handler function:

\[
\begin{align*}
\text{let} & \quad \text{run} \ f \ \text{init} = \\
\text{let} & \quad \text{exec} = \\
& \quad \text{match} \ f \ () \ \text{with} \\
& \quad \mid \ x \rightarrow (\text{fun} \ s \rightarrow (s, x)) \\
& \quad \mid \ \text{effect} \ (\text{Put} \ s') \ k \rightarrow (\text{fun} \ s \rightarrow \text{continue} \ k \ () \ s') \\
& \quad \mid \ \text{effect} \ \text{Get} \ k \rightarrow (\text{fun} \ s \rightarrow \text{continue} \ k \ s \ s)
\end{align*}
\]

in exec init
Evaluating an effectful program

```ocaml
define run f init =
define exec =
define 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 () ->
define let id = get () in
define let () = put (id + 1) in
  string_of_int id
) 3
```
Evaluating an effectful program

\[
\text{(match (fun () \rightarrow }
  \text{\quad let id = get () in}
  \text{\quad let () = put (id + 1) in}
  \text{\quad string_of_int id) ()}
\]

\[
\text{with}
\]
\[
| \quad x \rightarrow (\text{fun s \rightarrow (s, x)})
| \quad \text{effect (Put s') k \rightarrow (fun s \rightarrow continue k (()) s')}
| \quad \text{effect Get k \rightarrow (fun s \rightarrow continue k s s)}
\]

3
Evaluating an effectful program

```haskell
(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))
```
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

\[
\text{(fun } s \rightarrow \text{ continue } k s s) \ 3
\]
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 `>>=`
“Unnecessary” binds

The monad laws tell us that the following are equivalent:

\[
\begin{align*}
\text{return } v & \equiv k v \\
 v & \equiv \text{return } v
\end{align*}
\]

Why would we ever write the lhs?

“Administrative” \(\equiv\) and \text{return} arise through \textit{abstraction}

\[
\begin{align*}
\text{let } & \text{apply } f x = f \gg= \text{fun } g \rightarrow \\
& x \gg= \text{fun } y \rightarrow \\
& \text{return } (g y)
\end{align*}
\]

... 

\text{apply } (\text{return } \text{suc}) y 
\text{(* used: two returns, two }\gg=\text{ s *)} \\
\text{(* needed: one return, one }\gg=\text{ *)}
Effects from monads: the elements

```ocaml
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

```ocaml
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

```ocaml
module StateR = RR(State)

Build effectful functions from primitive effects get, put:

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

Build the handler from reify and State.run:

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

Use `let` instead of `>>=`:

```ocaml
code
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 $\Rightarrow$)

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