Recap: effects

Effects

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

Performing effects

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

Handling effects

\[
\text{match e with}
\]

\[
\ldots
\]

\[
| \text{effect (E x) k } \to \ldots
\]

Running continuations

\[
\text{val continue : ('a, 'b) continuation } \to 'a \to 'b
\]
Recap: state as a monad

The type of computations:

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

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

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

Signatures of primitive effects:

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

Primitive effects and a `run` function:

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

Primitive effects:

```ocaml
effect Put : state \rightarrow unit
effect Get : state
```

Functions to perform effects:

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

A handler function:

```ocaml
let run f init =
  let exec =
    match f () with
    | x \rightarrow (fun s \rightarrow (s, x))
    | effect (Put s') k \rightarrow (fun s \rightarrow continue k () s')
    | effect Get k \rightarrow (fun s \rightarrow continue k s s)
  in exec init
```
Evaluating an effectful program

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

run (fun () ->
  let id = get () in
  let () = put (id + 1) in
  string_of_int id)
```

Evaluating an effectful program

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

3
```
Evaluating an effectful program

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

3
Evaluating an effectful program

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

3
Evaluating an effectful program

\[(\text{fun } s \rightarrow \text{continue } k \ s \ s) \ 3\]
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

\[
\text{(match (let () = put (3 + 1) in }
\text{  string_of_int 3)}
\text{ with}
\text{  | x \rightarrow (fun s \rightarrow (s, x))}
\text{  | effect (Put s’) k \rightarrow (fun s \rightarrow continue k () s’)}
\text{  | effect Get k \rightarrow (fun s \rightarrow continue k s s)) 3}
\]
Evaluating an effectful program

\[
\text{(match (let () = perform (Put 4) in string_of_int 3) with}
\]
\[
| \ x \rightarrow (\text{fun } s \rightarrow (s, x))
| \ \text{effect} \ (\text{Put } s') k \rightarrow (\text{fun } s \rightarrow \text{continue } k () s')
| \ \text{effect} \ \text{Get } k \rightarrow (\text{fun } s \rightarrow \text{continue } k s s)) \ 3
\]
Evaluating an effectful program

\[
\text{(fun } s \to \text{ continue } k () 4) \ 3
\]

\[
k = \text{(match } (\text{let } () = - \text{ in }) \\
\text{string_of_int 3)} \text{ with} \\
| x \to (\text{fun } s \to (s, x)) \\
| \text{effect } \text{(Put } s') k \to (\text{fun } s \to \text{ continue } k () s') \\
| \text{effect } \text{Get } k \to (\text{fun } s \to \text{ continue } k s s))
\]
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))
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

\[(\text{fun } s \rightarrow (s, "3")) \ 4\]
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:

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

Why would we ever write the lhs?

“Administrative” \(\gg=\) and \text{return} arise through abstraction

```ocaml
let apply f x = f >>= fun g \to
    x >>= fun y \to
    return (g y)
```

```
apply (return succ) y
(* used: two returns, two \(\gg=\)s *)
(* needed: one return, one \(\gg=\) *)
```
module type MONAD = sig
  type '_t
  val return : 'a → 'a _t
  val bind : 'a _t → ('a → 'b _t) → 'b _t
end

Given M : MONAD:

  effect E : 'a M.t → 'a

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

let reflect m = perform (E m)
Effects from monads: the functor

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

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

  let reflect m = perform (E m)
end
Example: state effect from the state monad

```ocaml
module StateR = RR(State)

Build effectful functions from primitive effects get, put:

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

Build the handler from reify and State.run:

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

Use let instead of >>=:

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

Applicatives are a weaker, more general interface to effects
($\otimes$ 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