Recap: effects

Effects

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

Performing effects

```
val perform : 'a eff → 'a
```

Handling effects

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

Running continuations

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

Recap: state as a monad

The type of computations:

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

The `return` and $\gg=$ functions from `MONAD`:

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

Signatures of primitive effects:

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

Primitive effects and a *run* function:

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

Example: state as an effect

Primitive effects:

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

Functions to perform effects:

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

A handler function:

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

Evaluating an effectful program

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


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

Evaluating an effectful program

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

Evaluating an effectful program

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

Evaluating an effectful program

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

Evaluating an effectful program

```
(fun s → continue k s s) 3
```

Evaluating an effectful program

```
continue k 3 3

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

Evaluating an effectful program

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

Evaluating an effectful program

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

Evaluating an effectful program

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

Evaluating an effectful program

```
 (fun s → continue k () 4) 3

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

Evaluating an effectful program

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

Evaluating an effectful program

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

Evaluating an effectful program

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

Evaluating an effectful program

```
(fun s → (s, "3")) 4
```

Evaluating an effectful program

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

Effects and monads

Integrating effects and monads

What we'll get

Easy reuse of existing monadic code

(Uniformly turn monads into effects)

Improved efficiency, eliminating unnecessary binds

(Normalize computations before running them)

No need to write in monadic style

Use `let` instead of $\gg=$

"Unnecessary" binds

The monad laws tell us that the following are equivalent:

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

Why would we ever write the lhs?

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

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

Effects from monads: the elements

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

Given `M : MONAD`:

```
  effect E : 'a M.t → 'a


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


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

Effects from monads: the functor

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

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

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

Example: state effect from the state monad

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

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

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

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

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

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

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

Summary

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

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

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

We can build effects using handlers

Existing monads transfer uniformly

Next time: multi-stage programming

.< e >.