Effects

February 2018

Last time: monads and applicatives

$$\gg\!=$$

$$\otimes$$

This time: algebraic effects

```
effect E
```

monads

```
let x₁ = e₁ in
let x₂ = e₂ in
   ...
let xₙ = eₙ in
  e
```

applicatives

```
let x₁ = e₁
and x₂ = e₂
   ...
and xₙ = eₙ in
  e
```

Algebraic effects and handlers

(`effect E`)

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x -> g x
| B y -> h y
| ...
```

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x -> g x
| B y -> h y
| ...
```

`f v` *evaluates to the **value*** `A x`: *evaluate* `g x`

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x -> g x
| B y -> h y
| ...
```

`f v` *evaluates to the* ***value*** `A x`: *evaluate* `g x`

`f v` *evaluates to the* ***value*** `B y`: *evaluate* `h y`

...

Extending `match` for exceptions

Possible outcomes of `match`

```
match f v with
| A x -> g x
| B y -> h y
| ...
```

`f v` *evaluates to the **value*** `A x`: *evaluate* `g x`

`f v` *evaluates to the **value*** `B y`: *evaluate* `h y`

...

`f v` *raises an **exception*** `E`: `raise E`

Extending match for exceptions

New syntax:

```
match f v with
| A x -> g x
| B y -> h y
| ...
| exception (E z) -> j z
```

E.g. search an association list `l` for a boolean value:

```
match List.assoc s l with
| true ->  "found (True)"
| false -> "found (False)"
| exception Not_found -> "not found"
```

Extending `match for effects`

Possible outcomes of `match`

```
match f v with
| A x -> g x
| B y -> h y
| exception (E z) -> j z
| ...
```

`f v` *evaluates to the* ***value*** `A x`: *evaluate* `g x`

`f v` *evaluates to the* ***value*** `B y`: *evaluate* `h y`

`f v` *raises an* ***exception*** `E`: `raise E`

...

Extending `match` for effects

Possible outcomes of `match`

```
match f v with
| A x -> g x
| B y -> h y
| exception (E z) -> j z
| ...
```

`f v` *evaluates to the **value*** `A x`: *evaluate* `g x`

`f v` *evaluates to the **value*** `B y`: *evaluate* `h y`

`f v` *raises an **exception*** `E`: `raise E`

...

`f v` *performs an **effect*** `E` *and continues*: `perform E`, `continue`

Extending `match` for effects

New syntax:

```
match f v with
| A x -> g x
| B y -> h y
| ...
| effect (E z) k -> j z k
```

E.g. log each key while searching an association list `l`:

```
match List.assoc s l with
| true ->  "found (True)"
| false -> "found (False)"
| effect (Log key) k -> print key; continue k ()
| exception Not_found -> "not found"
```

Defining

```
type 'a t = ..
```

Extending

```
type 'a t +=
    G : int t
  | P : int → unit t
```

Constructing

```
P 3  (* No different to standard variants *)
```

Matching

```
let f : type a. a t → string = function
    G   → "G"
  | P _ → "P"
  | _   → "?"  (* All matches must be open *)
```

Elements of exceptions

Exceptions

`exception E: s -> exn` (*means:* `type exn += E: s -> exn`)

Raising exceptions

```
val raise : exn -> 'b
```

Handling exceptions

```
match e with
...
| exception (E x) -> ...
```

Running continuations

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
```

Using effects: yet another OCaml fork

modular implicits

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

effects

```
opam switch 4.03.0+effects
```

staging (final weeks)

```
opam switch 4.03.0+effects-ber
```

Example: exceptions as an effect

Define the effect and a function to **perform the effect**:

```
effect Raise : exn -> 'a
let raise e = perform (Raise e)
```

Define a function to **handle the effect**:

```
let _try_ f handle =
 match f () with
 | v -> v
 | effect (Raise e) k -> (* discard k! *) handle e
```

Program in **direct** (non-monadic) **style**:

```
let rec assoc x = function
| [] -> raise Not_found
| (k,v)::t -> if k = x then v else assoc x t

_try_ (fun () -> Some (assoc 3 l))
      (fun ex -> None)
```

Recap: state as a monad

The type of computations:

```
type 'a t = state -> state * 'a
```

The return and ≫= 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
```

The handler function for state:

```
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
```

Running the counter program under the state handler:

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

Starting point: reduce the function application

```
(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

Call the get function

```
(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

Perform the Get effect

```
(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

Evaluate the right-hand side of the case for `effect Get`

```
(fun s -> continue k s s) 3
```

Evaluating an effectful program

Evaluate the right-hand side of the case for `effect Get`

```
(fun s -> continue k s s) 3
```

(But what is `k`?)

Evaluating an effectful program

```
continue k 3 3
```

`k` is the program (up to the handler) with a hole for `perform Get`:

```
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))
```

hole

Evaluating an effectful program

```
continue k 3 3
```

value for hole

`k` is the program (up to the handler) with a hole for `perform Get`:

```
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))
```

hole

Evaluating an effectful program

Fill the hole and continue:

```
(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

Call the put function

```
(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

Perform the Put effect

```
(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))
```

hole

Evaluating an effectful program

value for hole

```
 (fun s -> continue k () 4) 3
```

hole

```
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))
```

(No more effects: evaluation continues as normal)

```
(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

(No more effects: evaluation continues as normal)

```
(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

(No more effects: evaluation continues as normal)

```
(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

(No more effects: evaluation continues as normal)

```
(fun s -> (s, "3")) 4
```

Evaluating an effectful program

(No more effects: evaluation continues as normal)

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

What to do with a continuation?

What can we do with the continuation `k`? Some possibilities:

Discard it (exceptions)

Call it (e.g. state)

Return it, and call it later (coroutines, generators)

Switch between continuations (concurrency)

Example: traversing trees in OCaml

A tree traversal:

```
let rec iter_tree f = function
 | Empty -> ()
 | Tree (l, x, r) -> iter_tree f l; f x; iter_tree f r
```

Using current OCaml's built-in (non-algebraic) effects, we can . . .

. . . **end** the computation **early** (using **exceptions**):

```
iter_tree (fun x -> if x = 0 then raise Zero)
```

. . . **accumulate** information (using **state**):

```
iter_tree (fun x -> sum := !sum + x)
```

Can we **pause** the traversal and **resume** it later**?**

Example: traversing trees with algebraic effects

Define a data type to represent the state of a traversal:

```
type 'a next =
   End : 'a next
 | Value : 'a * (unit -> 'a next) -> 'a next
```

Define an effect `Next` that carries values (tree labels):

```
effect Next : int -> unit
let next v = perform (Next v)
```

Handle `Next` by returning the continuation

```
let generate t =
 match iter_tree next t with
 | () -> End
 | effect (Next v) k -> Value (v, fun () -> continue k ())
```

Example: traversing trees with algebraic effects

```
let t = Tree (Tree (Empty, 3, Empty),
              4,
              Tree (Empty, 5, Empty))
```

generate t ⇝ Next (3, $\texttt{next}_1$)

$\texttt{next}_1$ () ⇝ Next (4, $\texttt{next}_2$)

$\texttt{next}_2$ () ⇝ Next (5, $\texttt{next}_3$)

$\texttt{next}_3$ () ⇝ End

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:

$$\text{return v} \gg= \text{k} \quad \equiv \quad \text{k v}$$
$$\text{v} \gg= \text{return} \quad \equiv \quad \text{v}$$

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`, define an effect and two conversions:

```
effect E : 'a M.t -> 'a
```

`reify` turns a function into a monadic computation

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

`reflect` turns a monadic computation into a function

```
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
```

Run the program using reify and `State.run`:

```
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
(⊗ 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: inductive families

```
Vec : Set → ℕ → Set
```