Algebraic Effects and Handlers

Andrej Bauer
University of Ljubljana

July 21, 2018

1 Programming with Algebraic Effects and Handlers

1.1 Effects

```
effect Abort : unit -> empty

let example b =
  handle
    let x = 7 in
    let y = 8 in
    if b then
      (match perform (Abort ()) with)
      **Without match/with get type error
    else
      x + y
  with
  | v -> v + 20
  | effect (Abort ()) k -> 42 **Without this get uncaught exception
```

Can also write as:

```
.
.
.
    if b then
      let result = perform(Abort ()) in
      (match result with)
.
.
.
```

1.2 State

Remember: $\texttt{Free}_{\texttt{State}}(V) \cong S \to S \times V$
Turn this into $\texttt{Tree}_{\texttt{State}}(V)$

```
(** State *)

effect Get: unit -> int
effect Set: int -> unit

(* The standard state handler. *)
let state' = handler
  | v -> (fun _ -> v)
  | effect (Get ()) k -> (fun s -> k s s)
  | effect (Set s') k -> (fun _ -> k () s')
;;
** state' : 'a => int -> 'a = <handler>

let example1 () =
  let f =
    (with state' handle
      let x = perform (Get ()) in
      perform (Set (2 * x)) ;
      perform (Get ()) + 10)
  in
    f 30
;;
** example1 : unit -> int -> int = <fun>

(* Better state handler, using finally clause *)
let state initial = handler
  | y -> (fun _ -> y)
  | effect (Get ()) k -> (fun s -> k s s)
  | effect (Set s') k -> (fun _ -> k () s')
  | finally f -> f initial
    **Add this to apply instead of returning a function
;;

let example2 () =
  with state 30 handle **Can call with initial state (30)
    let x = perform (Get ()) in
    perform (Set (2 * x)) ;
    perform (Get ()) + 10
```

1.3 Different handler types

exception-like: don't invoke continuation (must be deleted manually?) single-shot: calls continuation once (optimized in multicore ocaml) multi-shot: calls continuation multiple times (must be explicitly copied)

1.4 Ambivalent Choice

Introduced two funcitons:

```
Fail : unit -> empty
Choose : 'a list -> a
```

Where choose doesn't fail whenever possible.

1.4.1 Queen's Problem

```
(* The queens problem using ambivalent choice. *)

type queen = int * int

effect Select : int list -> int
effect Fail : unit -> empty

(* Do the given queens attack each other? *)
let no_attack (x,y) (x',y') =
  x <> x' && y <> y' && abs (x - x') <> abs (y - y')
;;

(* Given that queens qs are already placed, return the list of
rows in column x which are not attacked yet. *)
let available x qs =
  filter (fun y -> forall (no_attack (x,y)) qs) [1;2;3;4;5;6;7;8]
;;

(* Solve the queens problem by guessing what to do *)
let queens () =
  let rec place x qs =
    if x = 9 then
      qs
    else
      let y = perform (Select (available x qs)) in
      place (x+1) ((x,y) :: qs)
  in
    place 1 []

(* A handler for ambivalent choice which uses depth-first search *)
```

```
let dfs = handler
  | v -> v
  | effect (Select lst) k ->
    let rec tryem = function **Recursive function
      | [] -> (match perform (Fail ()) with)
      | x::xs -> (handle k x with effect (Fail ()) _ -> tryem xs)
        Try, and if fail, then handle that and try the next
    in
      tryem lst
;;

(* And we can solve the problem: *)
let solution =
  with dfs handle queens ()
;;
```

Create handler that finds all solutions:

```
let dfs_all = handler
  | v -> [v]
  | effect (Select lst) k ->
    let rec tryem = function
      | [] -> []
      | x::xs -> (handle k x with
                  | lst -> lst @ (tryem xs)
                  | effect (Fail ()) _ -> tryem xs)
```

1.5 Threads

1.5.1 Cooperative Multi-threading

See GitHub for code
He went a bit too quickly for me to follow exactly

```
type thread = unit -> unit
effect Yield : thread
effect Spawn : thread -> unit **Different from fork

(* We will need a queue to keep track of inactive threads.
We implement the queue as state. *)

effect Dequeue : unit -> thread option
effect Enqueue : thread -> unit
```

Queue:
Dequeue: if nothing then k, else k head of queue
Enqueue: just add k to the queue
Round robin:
How to dequeue a thread: Perform the operation dequeue, if get nothing then

done, else activate it
Yield: enqueue then dequeue
Spawn: enqueue the continuation then run t, but need to wrap t in the same handler.

1.6 Tree Rep. of a Functional

** Again...GitHub...**
Inverse to the function found on GitHub (i.e.: given h find the tree):

**Need a spy

```
effect Report : int -> bool

(** Convert a functional to a tree. *)
let rec fun2tree h =
  handle
    Answer (h (fun x -> perform (Report x)))
  with
    | effect (Report x) k -> Question (x, k false, k true)

let example1 = fun2tree (fun f -> true)
(* true *)

let example2 = fun2tree (fun f -> f 10; true)
(* Question (10, Answer true, Answer true) *)

let example3 = fun2tree (fun f -> if f 10 then (f 30 || f 15)
                                  else (f 20 && not (f 8)))
(* Question (10,
             Question (20, Answer false,
                       Question (8, Answer true, Answer false)),
             Question (30,
                       Question (15, Answer false, Answer true),
                       Answer true)) *)
```