Effective I/O

an unpromising approach to systems programming

Stephen Dolan
KC Sivaramakrishnan

OCaml Labs

Effect handlers

```
let do_stuff x =
  if x.is_bad then
    log “oh no”;
  process x
```

Effect handlers

```
let do_stuff x =
  if x.is_bad then
    raise PrettyBad;
  process x
```

```
match ... with
| result -> result
| exception PrettyBad ->
  log “it's pretty bad”;
  exit 1
```

Effect handlers

```
let do_stuff x =
  if x.is_bad then
    raise PrettyBad;
  process x
```

```
match ... with
| result -> result
| exception PrettyBad ->
 (* wish it kept going *)
```

Effect handlers

```
let do_stuff x =
  if x.is_bad then
    raise PrettyBad;
  process x
```

```
match ... with
| result -> result
| exception PrettyBad ->
  continue
```

Effect handlers

```
let do_stuff x =
  if x.is_bad then
    perform PrettyBad;
  process x
```

```
match ... with
| result -> result
| effect PrettyBad k ->
  continue k
```

Scheduling tasks

```
let run_q =
  Queue.create ()

let enqueue k =
  Queue.push k run_q

let rec dequeue () =
  if Queue.is_empty run_q then ()
  else continue (Queue.pop run_q) ()
```

Scheduling tasks

```
effect Yield : unit
effect Fork : (unit -> unit) -> unit

let rec schedule f =
  match f () with
  | () -> dequeue ()
  | effect Yield k ->
      enqueue k; dequeue ()
  | effect (Fork f) k ->
      enqueue k; schedule f
```

Direct-style I/O

```
let copy_channels =
  let buf_len = 65536 in
  let buf = Bytes.create buf_len in
  let rec loop ic oc =
    match input ic buf 0 buf_len with
    | 0 -> ()
    | n -> output oc buf 0 n; loop ic oc
  in
  loop
```

An effective stdlib

```
type in_channel = Reader.t

effect Input :
  in_channel * bytes * int * int
  -> int

let input ic buf pos len =
  perform (Input (ic, buf, pos, len))
```

Direct-style with Async

```
val run : (unit -> 'a) -> 'a Deferred.t

let run f =
  match f () with
  | x -> return x
  | effect (Input (ic, buf, pos, len)) k ->
     Reader.read ic buf ~pos ~len
       >>= fun x -> continue k x
```

Direct-style I/O

- Simple I/O interfaces use direct calls
- Efficient ones use callbacks, for overlapping
- With effects, we can write the simple code but run the fast code

Mixing styles

- Effects let us mix direct and monadic code
- Parts of the code can choose whether they need to control scheduler interactions
- Libraries can expose code without imposing a particular I/O style

Managing resources

```
let file = open_in “words.txt” in
match parse_contents file with
| result ->
  close file;
  result
| exception e ->
  close file;
  raise e
```

Managing resources

- Computations holding resources are linear
  - so their continuations are too!
- Linear continuations are very, very fast
- Lacking linear types, we fake them dynamically

Blocking I/O

- The operating system provides select(), telling us whether I/O will block
- But it lies, and it lies, and it lies.
- Async uses thread pools to deal with this

Thread pools vs. effects

- For operations which happen not to block, thread pools have high overhead
- With effects, we don't have to decide in advance whether to switch to another thread
- We can invoke continuations on any C thread

Overlapping with effects

```
| effect (Delayed id) k ->
  Hashtbl.add ongoing_io id k

| effect (Completed id) k ->
  let k' = Hashtbl.find ongoing_io id in
  Hashtbl.remove ongoing_io id;
  enqueue (fun () -> continue k ());
  continue k' ()
```

Questions?