CS 3110

Async

Prof. Clarkson
Fall 2015

Today's music: *It's Gonna be Me* by *NSYNC

Review

Previously in 3110

- Threads
- Async (futures): deferreds, scheduler, callbacks with upon and bind

Today:

- more on bind
- other sequencing operators
- ivars

Review: Async

- A third-party library for futures in OCaml
- **Deferreds**: values whose completed computation has been deferred until the future (and in fact is happening already)
- Scheduler runs **callbacks** that have been registered to consume the values of deferreds
  - Scheduler selects a callback whose input has become ready to consume,
  - runs the callback with that input,
    - Only ever one callback running at a time
    - Scheduler never interrupts the callback
  - and repeats.

Review: `Deferred`

An `'a Deferred.t` is like a box:

- It starts out empty
- At some point in the future, it could be filled with a value of type `'a`
- Once it's filled, the box's contents can never be changed ("write once")

Terminology:

- "box is filled" = "deferred is determined"
- "box is empty" = "deferred is undetermined"

Review: Registering a callback

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

- use to register a callback (the function of type `'a -> unit`) to run sometime after deferred is determined
- **`upon`** returns immediately with `()` no matter what
- sometime after box is filled (if ever), scheduler runs callback on contents of box
- callback produces `()` as return value, but never returned to anywhere

Question

Suppose you create a deferred with **`return 42`**. When is that deferred determined?

A. Immediately

B. At some point in the future, but you don't know when.

C. After the creator's callback returns control to the scheduler.

D. Never

E. None of the above

Question

Suppose you create a deferred with **`return 42`**. When is that deferred determined?

A. **Immediately**

B. At some point in the future, but you don't know when.

C. After the creator's callback returns control to the scheduler.

D. Never

E. None of the above

Review: Creating deferreds

```
return : 'a -> 'a Deferred.t
```

- use to create a deferred that is already determined

```
after : Core.Std.Time.Span.t
          -> unit Deferred.t
```

- use to create a deferred that becomes determined sometime after a given length of time
- `Core.Std.Time.Span.of_int_sec 10` represents 10 seconds

BIND

Bind

```
bind :
        'a Deferred.t
        -> ('a -> 'b Deferred.t)
        -> 'b Deferred.t
```

- use to register a deferred computation after an existing one
- takes two inputs: a deferred `d`, and callback `c`
- `bind d c` immediately returns with a new deferred `d'`
- sometime after `d` is determined (if ever), scheduler runs `c` on contents of `d`
- `c` produces a new deferred, which if it ever becomes determined, also causes `d'` to be determined with same value

Bind

```
Deferred.bind
  (return 42)
  (fun n -> return (n+1))
```

- first argument is a deferred that is determined with value **42**
- second argument is a callback that takes an integer **n** and returns a deferred that is determined with value **n+1**
- **bind** immediately returns with an undetermined deferred **ud**
- scheduler, when it next gets to run, can notice that first argument is determined, and run callback
- callback gets **42** out of box, **binds** it to **n**, and returns a new deferred that is determined with value **43**
- scheduler can notice that output of callback has become determined, and make **ud** determined with same value

>>=

(>>=)

- infix operator version of **bind**
- **bind d c** is the same as **d >>= c**

```
Deferred.bind
  (return 42)
  (fun n -> return (n+1))
(* equiv. *)
return 42 >>= fun n ->
return (n+1)
```

>>=

```
open Async.Std
let sec n = Core.Std.Time.Span.of_int_sec n
let return_after v delay =
  after (sec delay) >>= fun () ->
  return v
let _ =
  (return_after "First timer elapsed\n" 5) >>= fun s ->
  print_string s;
  (return_after "Second timer elapsed\n" 3) >>= fun s ->
  print_string s;
  exit 0
let _ = print_string "Hello\n"
let _ = Scheduler.go ()
```

Question

```
let _ =
  (return_after "First timer elapsed\n" 5) >>= fun s ->
  print_string s;
  (return_after "Second timer elapsed\n" 3) >>= fun s ->
  print_string s;
  exit 0
let _ = print_string "Hello\n"
```

Which string will be printed first?

A. "First timer elapsed"

B. "Second timer elapsed"

C. "Hello"

Question

```
let _ =
  (return_after "First timer elapsed\n" 5) >>= fun s ->
  print_string s;
  (return_after "Second timer elapsed\n" 3) >>= fun s ->
  print_string s;
  exit 0
let _ = print_string "Hello\n"
```

Which string will be printed first?

A. "First timer elapsed"

B. "Second timer elapsed"

C. "Hello"

Question

```
let _ =
  (return_after "First timer elapsed\n" 5) >>= fun s ->
  print_string s;
  (return_after "Second timer elapsed\n" 3) >>= fun s ->
  print_string s;
  exit 0
let _ = print_string "Hello\n"
```

Which string will be printed second?

A. "First timer elapsed"

B. "Second timer elapsed"

C. "Hello"

Question

```
let _ =
  (return_after "First timer elapsed\n" 5) >>= fun s ->
  print_string s;
  (return_after "Second timer elapsed\n" 3) >>= fun s ->
  print_string s;
  exit 0
let _ = print_string "Hello\n"
```

Which string will be printed second?

A. "First timer elapsed"

B. "Second timer elapsed"

C. "Hello"

What if you wanted the answer to be B?

Concurrently

```
let t1 =
  return_after "First timer elapsed\n" 5 >>= fun s ->
  print_string s;
  return ()

let t2 =
  return_after "Second timer elapsed\n" 3 >>= fun s ->
  print_string s;
  return ()

let _ =
  t1 >>= fun () ->
  t2 >>= fun () ->
  exit 0
```

Now the "second" timer string would be printed before the "first"

MORE SEQUENCING OPERATORS

Map

```
map :
        'a Deferred.t
        -> ('a -> 'b)
        -> 'b Deferred.t
```

- takes two inputs: a deferred **d**, and a function **f**
- **map d f** immediately returns with a new deferred **d'**
- sometime after **d** is determined (if ever), scheduler runs **f** on contents of **d**, immediately yielding a new value **b**, and **d'** is immediately determined with that value
- has its own infix operator **(>>|)**

Map

```
let return_after v delay =
  after (sec delay) >>= fun () ->
  return v

let return_after' v delay =
  after (sec delay)
  >>| fun () -> v
```

...how might you implement **`map`**?

Map

```
let map (d: 'a Deferred.t)
  (f: 'a -> 'b) : 'b Deferred.t
=
  d >>= fun a ->
  return (f a)
```

Both

```
both :
        'a Deferred.t
         -> 'b Deferred.t
         -> ('a*'b) Deferred.t
```

- takes two inputs: a deferred **d1**, and a deferred **d2**
- **both d1 d2** immediately returns with a new deferred **d**
- sometime after both **d1** and **d2** are determined (if ever), **d** is determined with the pair of values from inside **d1** and **d2**

...how might you implement **both**?

Both

```
let both
  (d1: 'a Deferred.t)
  (d2: 'b Deferred.t)
  : ('a*'b) Deferred.t
=
  d1 >>= fun a ->
  d2 >>= fun b ->
  return (a,b)
```

Question

Does this implementation force the contents of `d1` to be computed before the contents of `d2`?

```
let both d1 d2 =
  d1 >>= fun a ->
  d2 >>= fun b ->
  return (a,b)
```

A. Yes

B. No

Question

Does this implementation force the contents of `d1` to be computed before the contents of `d2`?

```
let both d1 d2 =
  d1 >>= fun a ->
  d2 >>= fun b ->
  return (a,b)
```

A. Yes

B. No

Either

```
either :
      'a Deferred.t
       -> 'a Deferred.t
       -> 'a Deferred.t
```

- takes two inputs: a deferred **d1**, and a deferred **d2**
- **either d1 d2** immediately returns with a new deferred **d**
- sometime after at least one of **d1** and **d2** is determined (if ever), **d** is determined with the same value
- no guarantee about timing of **d1** vs **d2** : maybe **d1** becomes determined first with value **v1** , then **d2** with **v2** , then **d** with **d2**

...how might you implement **either**?

Either

```
let either
  (d1: 'a Deferred.t)
  (d2: 'a Deferred.t)
  : 'a Deferred.t
=
  failwith "You can't"
```

IVARS

`Ivar`

An `'a Ivar.t` is like a box:

- It starts out empty
- At some point in the future, it could be filled with a value of type `'a`
- Once it's filled, the box's contents can never be changed ("write once")
- You can fill the box

Ivar

- `create : unit -> 'a Ivar.t`
- `is_full  : 'a Ivar.t -> bool`
- `fill : 'a Ivar.t -> 'a -> unit`
  - Attempting to fill an already full ivar raises an exception
  - That's where the name comes from...

Digression on Cornell history

- i = incremental
- Originally [Arvind and Thomas 1981] *I-structures* were a kind of data structure for functional arrays in which each element could be assigned exactly once—hence the array was constructed *incrementally*
- Used for parallel computing in language called Id [Arvind, Nikhil, and Pingali 1986]
  - Keshav Pingali, Cornell CS prof 1986-2006?
- Implemented in *Concurrent ML* by John Reppy (Cornell PhD 1992)

Ivar

- `create : unit -> 'a Ivar.t`
- `is_full  : 'a Ivar.t -> bool`
- `fill : 'a Ivar.t -> 'a -> unit`
  - Attempting to fill an already full ivar raises an exception
  - That's where the name comes from

...but how can you get a value out of the ivar?

`Ivar`

```
read : 'a Ivar.t -> 'a Deferred.t
```

- `read i` immediately returns a deferred that becomes determined after `i` is filled
- and to get a value out of that deferred, use any of the ways we've seen of registering callbacks

now we can implement **`either`**...

Either

```
let either d1 d2 =
  let result = Ivar.create () in
  let fill = fun x ->
    if Ivar.is_empty result
    then Ivar.fill result x
    else () in
  upon d1 fill;
  upon d2 fill;
  Ivar.read result
```

Upcoming events

- [Thursday] A4 due
- [next Monday] project charter due

This is in sync.

THIS IS 3110