CS 3110

Futures

Prof. Clarkson
Fall 2016

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

Review

Previously in 3110:

- Functional programming
- Modular programming/software engineering
- Interpreters
- Reasoning about correctness

Final unit of course: Advanced topics

Today:

- Futures: a data structure and programming paradigm for concurrency
- Implementation in Jane Street's Async library

Concurrency

- Networks have multiple computers
- Computers have multiple processors
- Processors have multiple cores

...all working semi-independently
...all sharing resources

concurrent: overlapping in duration
sequential: non-overlapping in duration
parallel: happening at the same time

Concurrency

At any given time, my laptop is...

- Streaming music
- Running a web server
- Syncing with web services
- Scanning for viruses
- Running OCaml

The OS plays a big role in making it look like those all happen simultaneously

Concurrency

Applications might also want concurrency:

- Web server that handles many clients at once
- Scientific calculations that exploit parallel architecture to get speedup
- Simulations that model physical processes
- GUIs that want to respond to users while doing computation (e.g., rendering) in the background

Programming models for concurrency

Threads: sequential code for computation
e.g., Pthreads, OpenMP, java.lang.Thread
OCaml `Thread`

Futures: values that are maybe not yet computed
e.g., .NET async/await, Clojure, Scala, java.util.concurrent.Future
OCaml `Async` and `Lwt`

(and many others)

FUTURES

Futures

- *Future*: computation that will produce a value sometime in the future
  - aka *promises* or *delays*
- Completion of computation:
  - *implicit:* when used, computation forced to occur
  - *explicit:* call a function to force computation
- Initiation of computation:
  - *eager:* starts right away
  - *lazy*: starts only when needed

Async

- A third-party library for futures in OCaml
  - To install: `opam install async` (will take a long time)
- Instead of "futures" calls the abstraction **deferreds**, as in *values whose completed computation has been deferred until the future (and in fact is happening already)*
- Typical use of library is to do asynchronous I/O
  - Launch an I/O operation as a deferred
  - Later on its results will be available
  - Enables latency hiding: have multiple I/O operations occurring in parallel

(A)synchronous I/O

- *Synchronous* aka *blocking* I/O:
  - call I/O function which *blocks*, wait for completion...
  - then continue your computation
  - e.g., `Pervasives.input_line : in_channel -> string`
- *Asynchronous* aka *non-blocking* I/O:
  - call I/O function which is *non-blocking*, function immediately returns, continue your computation, later...
  - I/O completes
  - e.g., `Async.Std.Reader.file_contents : string -> string Deferred.t`
  - how does program make use of completed I/O? ...

Async: Print file length

```
open Async.Std

let println s =
  printf "%i\n" (String.length s)

let r = Reader.file_contents Sys.argv.(1)
let _ = upon r (fun s -> println s; ignore(exit 0))

let _ = Scheduler.go()
```

To compile: `corebuild -pkg async` ***`filename`*****`.byte`**

Scheduler

- Scheduler runs **callbacks** that have been registered to consume the values of deferreds
- Only ever one callback running at a time
  - Async is "single threaded"
  - No true parallelism: designed for latency hiding not parallel speedup
  - The OCaml runtime itself is single threaded
- Scheduler:
  - selects a callback whose input has become ready to consume
  - runs the callback with that input
  - never interrupts the callback
    - if callback never returns, scheduler never gets to run again!
    - **cooperative** concurrency
  - repeats

Deferred so far

```
module Async.Std : sig
  val upon : 'a Deferred.t -> ('a -> unit) -> unit

  module Deferred : sig
    type 'a t
    ...
  end

  module Reader : sig
    val file_contents : string -> string Deferred.t
    ...
  end

  ...
end
```

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"

Manipulating boxes

`peek : 'a Deferred.t -> 'a option`

- use to see whether box has been filled yet
- returns immediately with **`None`** if nothing in box
- returns immediately with **`Some a`** if **`a`** is in box

Manipulating boxes

```
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

Creating boxes

```
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.sec 10.0` represents 10.0 seconds and has type `Core.Std.Time.Span.t`

Creating boxes

- `file_contents : string -> string Deferred.t`
  - use to read entire contents of file into a string
  - `file_contents` returns immediately with an empty deferred
  - program can now continue with doing other things (scheduling other I/O, processing completed I/O, etc.)
  - at some point in the future, when file read completes (if ever), that deferred becomes determined
  - any callbacks registered for the deferred will then (eventually) be executed with the deferred

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

BIND

Bind

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

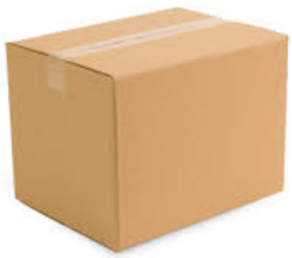
- use to register a deferred computation after an existing one
- takes two inputs: a deferred **`d`**, and callback **`c`**
- **`bind d c`** immediately returns with a new deferred **`d'`**
- sometime after **`d`** is determined (if ever), scheduler runs **`c`** on contents of **`d`**
- **`c`** produces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

IVARS

`Ivar`

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

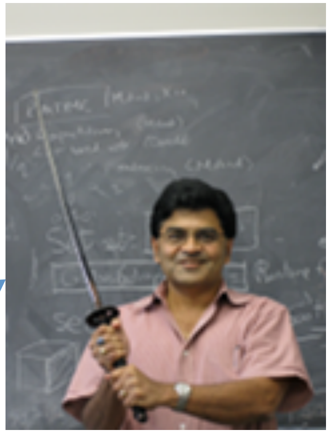
- It starts out empty
- At some point in the future, it could be filled with a value of type `'a`
- Once it's filled, the box's contents can never be changed ("write once")
- You can fill the box

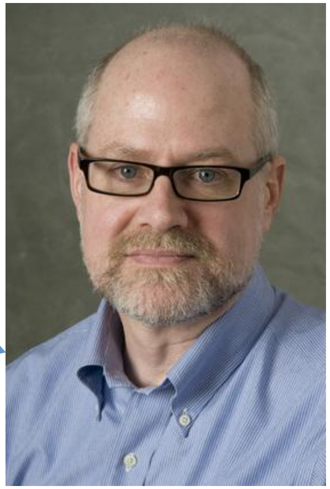
Ivar

- `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

Upcoming events

- [between now and next Thursday] MS1 design review meetings: **you** need to schedule with your grader
- [next Thursday] Prelim 2

This is in sync.

THIS IS 3110

MORE ABOUT BIND

>>=

```
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 without ivars"
```

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
```