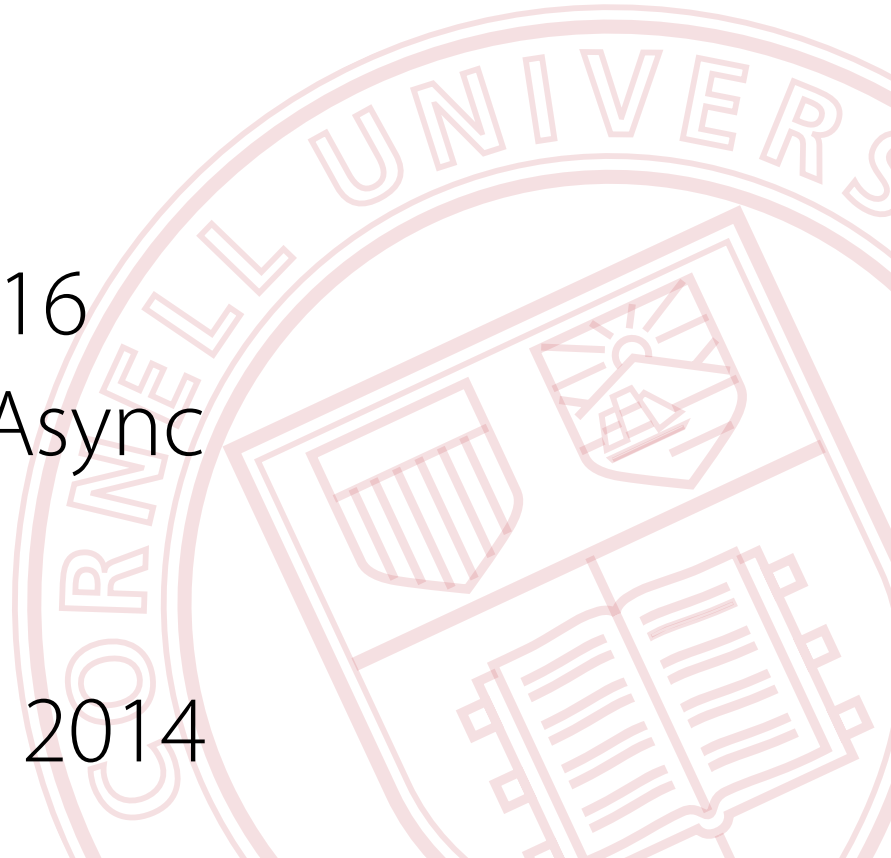
# CS 3110

# Data Structures and Functional Programming



Lecture 16

Advanced Async

23 October 2014

# Review

So far we've explored some basic examples of programming with *deferred computations* in Async...

# Review

So far we've explored some basic examples of programming with *deferred computations* in Async...

```
module Deferred : sig =
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val both : 'a t -> 'b t -> (a * b) 't
  ...
end
```

# Async Pipes

A common use of concurrency is to wrap I/O operations

`Pipe` defines abstractions for asynchronous I/O

# Async Pipes

A common use of concurrency is to wrap I/O operations

`Pipe` defines abstractions for asynchronous I/O

```
module Pipe : sig =
  val create : unit -> 'a Reader.t * 'a Writer.t
  val read : 'a Reader.t ->
             [ `Eof | `Ok of 'a ] Deferred.t
  val write : 'a Writer.t -> 'a -> unit Deferred.t
  val close : 'a Writer.t -> unit
  val close_read : 'a Reader.t -> unit
  val transfer : 'a Reader.t -> 'b Writer.t ->
                 f:('a -> 'b) -> unit Deferred.t
end
```

# Example: Echo Server

```
open Core.Std
open Async.Std

let run () =
  let handler _ (r:Reader.t) (w:Writer.t)
  : unit Deferred.t
  = Pipe.transfer  f:(fun x -> x)
  (Reader.pipe r) (Writer.pipe w) in
  let () =
    ignore
    (Tcp.Server.create
      (Tcp.on_port 3110)
      handler) in
  Deferred.never ()

let () =
  don't_wait_for (run () >>= fun () -> exit 0);
  ignore (Scheduler.go ())
```

# Synchronization and Coordination

Last time:

```
let both (d1:'a Deferred.t) (d2:'b Deferred.t)
  : ('a * 'b) Deferred.t
  = d1 >>= fun v1 ->
    d2 >>= fun v2 ->
    return (v1,v2)
```

# Synchronization and Coordination

## Last time:

```
let both (d1:'a Deferred.t) (d2:'b Deferred.t)
  : ('a * 'b) Deferred.t
  = d1 >>= fun v1 ->
    d2 >>= fun v2 ->
    return (v1,v2)
```

## Today:

```
module Deferred : sig =
  ...
  val all (l:'a Deferred.t list) -> ('a list) Deferred.t
  val any (l:'a Deferred.t list) -> 'a Deferred.t
  ...
end
```

# Timeouts

Suppose we want to execute a deferred computation either until it finishes or we run out of time...

```
val Core.Time.Span.of_sec : float -> Core.Span.t
val after : Core.Time.Span.t -> unit Deferred.t
```

# Timeouts

Suppose we want to execute a deferred computation either until it finishes or we run out of time...

```
val Core.Time.Span.of_sec : float -> Core.Span.t
val after : Core.Time.Span.t -> unit Deferred.t
```

Example: `after (Core.Time.Span.of_sec 1.0)` is determined after a second

# Timeouts

Suppose we want to execute a deferred computation either until it finishes or we run out of time...

```
val Core.Time.Span.of_sec : float -> Core.Span.t
val after : Core.Time.Span.t -> unit Deferred.t
```

Example: `after (Core.Time.Span.of_sec 1.0)` is determined after a second

We can define a general `timeout` function as follows:

```
let timeout (thunk:unit -> 'a Deferred.t) (n:float)
  : ('a option) Deferred.t
  = Deferred.any
  [ after (Core.Time.Span.of_sec n) >>| (fun () -> None);
    thunk () >>| (fun x -> Some x) ]
```

# Ivars

The operations **`both`**, **`any`**, and **`all`** suffice for the majority of situations that arise in practice...

...but sometimes we need finer-grained control.

```
module IVar : sig
  type 'a t
  val create : unit -> 'a t
  val fill : 'a t -> 'a -> unit
  val fill_if_empty : 'a t -> 'a -> unit
  val is_empty : 'a t -> bool
  val is_full : 'a t -> bool
  val read : 'a t -> 'a Deferred.t
  ...
end
```

# Delaying Scheduler

```
module type DELAYER = sig
  type t
  val create : float -> t
  val schedule : t -> (unit -> 'a Deferred.t) ->
                 'a Deferred.t
  end
```

# Delaying Scheduler

```
module type DELAYER = sig
  type t
  val create : float -> t
  val schedule : t -> (unit -> 'a Deferred.t) ->
                 'a Deferred.t
  end
```

The following function will be useful:

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

# Choice

The `any` function function provides a way to choose just one of several deferred computations.

But if those computations have effects, such as opening files or socket connections, printing to the console, etc., the behavior may not be what is desired.

```
module Deferred : sig =
  ...
  type choice
  val choice : 'a t -> ('a -> 'b) -> 'b choice
  val choose : ('a choice) list -> 'a t
  end
```

# Choice

The `any` function function provides a way to choose just one of several deferred computations.

But if those computations have effects, such as opening files or socket connections, printing to the console, etc., the behavior may not be what is desired.

```
module Deferred : sig =
  ...
  type choice
  val choice : 'a t -> ('a -> 'b) -> 'b choice
  val choose : ('a choice) list -> 'a t
  end
```

Question: now can you implement `any`?