CS 3110

Futures

Nate Foster
Spring 2018

Review

Previously in 3110:

- Functional programming
- Modular programming
- Interpreters
- Formal methods

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
- 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*
- Various designs:
  - Completion of computation can be...
    - **implicit:** when used, computation forced to occur
    - **explicit:** call a function to force computation
  - Initiation of computation can be...
    - **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.Reader.file_contents : string -> string Deferred.t`
  - how does program make use of completed I/O? ...

Async: Print file length

```
open Async

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 : 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's return value `()` never used by anyone

Creating boxes

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

– use to create a deferred that is already determined

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

– use to create a deferred that becomes determined sometime after a given length of time

– `Core.sec 10.0` represents 10.0 seconds and has type `Core.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

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

(>>=)

- 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)
```

Let Notation

`let%bind c = d`

- Let version of **`bind`**
- same as **`d >>= c`**
- Must use **`Let_syntax`**, compile with **`ppx_let`**

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

Upcoming events

- [by Friday] A5 released
- [Friday] Yaron Minsky on “Effective ML”
  - 5:30pm
  - Gates G01
  - Pizza!