

Concurrency in OCaml

Tobias Hoffmann

Why Concurrency?

Wait for...

- Network
- Filesystem
- User Input

How Concurrency?

- Threads (*Java, C#*)
 - Resource intensive
 - Requires locking of shared data
- Event Loop (*Javascript, GUI-Frameworks*)
 - Inverted control
- ✨ Hybrid Async ✨ (*OCaml*)

The Async Library in Ocaml

- User-level threads
- Non-Preemptive Scheduler
- In `utop` (repl):
 - Scheduler runs automatically
 - Block & wait upon evaluation of deferred variable

Normal File I/O

```
open Core;;

Out_channel.write_all "greeting.txt" ~data:"Hello World!";;
(* => () *)

In_channel.read_all "greeting.txt";;
(* => "Hello World!" *)
```

We have to wait for the read/write calls

Async File I/O

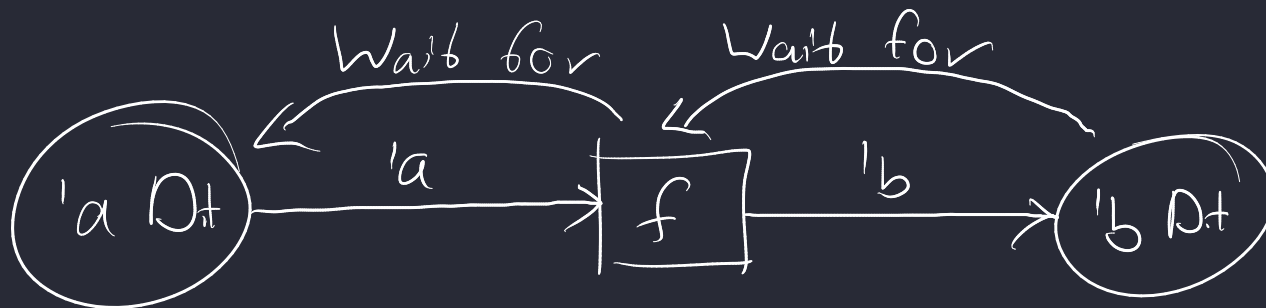
```
#require "async";;  
open Async;;  
  
let greeting = Reader.file_contents "greeting.txt";;  
(* : string Deferred.t *)  
  
Deferred.peek greeting;;  
(* => None *)  
  
greeting;;  
(* => "Hello World!" *)  
  
Deferred.peek greeting;;  
(* => Some "Hello World!" *)
```

Chaining Async Operations

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

Chain one async operation behind another

Once the `'a Deferred.t` becomes determined,
schedule `f` with it's contents as argument



Uppercase File with Bind

```
let greeting = Reader.file_contents "greeting.txt";;  
(* : string Deferred.t *)  
  
let write_result = Deferred.bind greeting ~f:(fun text ->  
    Writer.save "greeting.txt" ~contents:(String.uppercase text));;  
(* : unit Deferred.t *)  
  
In_channel.read_all "greeting.txt";;  
(* => "Hello World!" *)  
  
write_result;;  
(* => () *)  
  
In_channel.read_all "greeting.txt";;  
(* => "HELLO WORLD!" *)
```


Uppercase File as a Function

```
let uppercase_file fname =  
  Deferred.bind (Reader.file_contents fname) ~f:(fun text ->  
    Writer.save fname ~contents:(String.uppercase text));;  
(* : string -> unit Deferred.t *)  
  
let write_result = uppercase_file "greeting.txt";;  
(* : unit Deferred.t *)  
  
Deferred.peek write_result;;  
(* => None *)  
  
write_result;;  
  
Deferred.peek write_result;;  
(* => Some () *)
```

Aliases for Bind

Infix operator

```
let uppercase_file fname =  
  Reader.file_contents fname  
  >>= fun text ->  
    Writer.save fname ~contents:(String.uppercase text);;
```

Let Syntax

```
#require "ppx_let";;  
  
let uppercase_file fname =  
  let%bind text = Reader.file_contents fname in  
  Writer.save fname ~contents:(String.uppercase text);;
```

Create Deferred with Return

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

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

```
let deferred_int = return 1729;;  
(* : int Deferred.t *)
```

```
Deferred.peek deferred_int;;  
(* => Some 1729 *)
```

```
let count_lines fname =  
  Deferred.bind (Reader.file_contents fname)  
  ~f:(fun text ->  
    return (List.length (String.split text ~on:'\n')));;
```

Combine Bind and Return into Map

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

```
let count_lines fname =  
  Deferred.map (Reader.file_contents fname)  
    ~f:(fun text ->  
      List.length (String.split text ~on:'\n'));;
```

Aliases for Map

Infix operator

```
let count_lines fname =  
  Reader.file_contents fname  
  >>| fun text ->  
    List.length (String.split text ~on:'\n');
```

Let Syntax

```
#require "ppx_let";;  
  
let count_lines fname =  
  let%map text = Reader.file_contents fname in  
  List.length (String.split text ~on:'\n');
```

An Async Webserver

```
Tcp.Server.create : ... -> Tcp.Server.t Deferred.t
```

```
let server () =  
  let host_and_port =  
    Tcp.Server.create ~on_handler_error:`Raise  
      (Tcp.Where_to_listen.of_port 1729) (fun _ r w -> respond r w)  
  in  
  ignore host_and_port
```

```
ignore : 'a -> unit
```

The Response Function

```
let rec respond r w =  
  match%bind Reader.read_line r with  
  | `Eof -> return ()  
  | `Ok line ->  
    let%bind () = Writer.flushed w in  
    Writer.write_line w (String.uppercase line) ;  
    respond r w  
(* : Reader.t -> Writer.t -> unit Deferred.t *)
```

`match%bind` is analogous to `let%bind`

Starting the server

```
let () =  
  server () ;  
  never_returns (Scheduler.go ())
```

=> Demo!

Note: Enable `ppx_let` syntax extension in a dune project:

```
(preprocess  
  (pps ppx_let))
```


Monads

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> f:('a -> 'b t) -> 'b t
end;;

module _ : Monad = Deferred;;
```

- A common pattern in functional programming
- Represent chainable operations with context
- Related: Applicative , Functor (-> `map`)

Option as a Monad

```
module Option : Monad with type 'a t = 'a option = struct
  type 'a t = 'a option

  let return x = Some x

  let bind m ~f =
    match m with
    | None -> None
    | Some x -> f x
end;;
```

Chaining Option Monads

```
let half x =  
  if x mod 2 = 0 then  
    Some (x / 2)  
  else  
    None  
  
let m1 = Some 6;;  
  
let m2 = Option.bind m1 ~f:half;;  
(* => Some 3 *)  
  
let m3 = Option.bind m2 ~f:half;;  
(* => None *)  
  
let m4 = Option.bind m3 ~f:half;;  
(* => None *)
```

Conclusion

- Async values are wrapped in `Deferred.t`
 - Chain async operations with `Deferred.bind` & `Deferred.map`
 - Don't forget to start the scheduler!
-
- More complex async operations with `Async.Ivar`
 - ⚠ Exception handling with Async
 - True multithreading with OCaml 5.0 (2022)