Discussion 15

Promises

Kenneth Fang (kwf37), Newton Ni (cn279)

March 20, 2019

Key Ideas

- Like laziness: do it **eventually**

Key Ideas

- Like laziness: do it **eventually**
- Bind is your best friend: `'a t -> ('a -> 'b t) -> 'b t`

Why Promises?

- Very useful for IO-bound computations and message-passing

Why Promises?

- Very useful for IO-bound computations and message-passing
- Don't just sit around waiting

Why Promises?

- Very useful for IO-bound computations and message-passing
- Don't just sit around waiting
- Push vs. poll architecture

Why Promises?

- Very useful for IO-bound computations and message-passing
- Don't just sit around waiting
- Push vs. poll architecture
- Not useful by themselves: need OS support

Bind

- Pipelining should be familiar by now: |>

Bind

- Pipelining should be familiar by now: `|>`
- `(|>) : 'a -> ('a -> 'b) -> 'b`

Bind

- Pipelining should be familiar by now: `|>`
- `(|>) : 'a -> ('a -> 'b) -> 'b`
- Bind is just a fancy pipeline

Bind

- Pipelining should be familiar by now: `|>`
- `(|>) : 'a -> ('a -> 'b) -> 'b`
- Bind is just a fancy pipeline
- `(>>=) : 'a t -> ('a -> 'b t) -> 'b t`

Bind

- Pipelining should be familiar by now: `|>`
- `(|>) : 'a -> ('a -> 'b) -> 'b`
- Bind is just a fancy pipeline
- `(>>=) : 'a t -> ('a -> 'b t) -> 'b t`
- Both abstract over **sequential** composition

Bind: Function Syntax

Does this remind you of anything?

```
bind promise_a (
    fun a -> bind promise_b (
        fun b -> bind promise_c (
            fun c -> do_something_with a b c
        )
    )
)
```

Application: Function Syntax

```
map_refl refl (
    map_rotors_r_to_l rotors (
        map_plug plugboard (
            index c
        )
    )
)
```

Application: Operator Syntax

```
c |> index
  |> map_plug plugboard
  |> map_rotors_r_to_l rotors
  |> map_refl refl
```

Bind: Operator Syntax

```
promise_a >>= fun a ->
promise_b >>= fun b ->
promise_c >>= fun c ->
do_something_with a b c
```

Bind: PPX Extensions

- We don't see PPX extensions in this course for the most part

Bind: PPX Extensions

- We don't see PPX extensions in this course for the most part
- Compile-time code generation!

Bind: PPX Extensions

- We don't see PPX extensions in this course for the most part
- Compile-time code generation!
- Textbook section 8.20 for the curious

Bind: PPX Extensions

- We don't see PPX extensions in this course for the most part
- Compile-time code generation!
- Textbook section 8.20 for the curious
- Sneak peek:

```
let/lwt a = promise_a in let/lwt b = promise_b in do_sc
```

Libraries

- `##require "lwt";;` - core modules and types

Libraries

- `##require "lwt";;` - core modules and types
- `##require "lwt.unix";;` - Unix bindings for socket/file IO

Libraries

- `##require "lwt";;` - core modules and types
- `##require "lwt.unix";;` - Unix bindings for socket/file IO
- `UTop.set_auto_run_lwt false;;`
- Special utop behavior for top-level promises (see textbook 8.19)

Examples: Sequential Composition

```
let a () = Lwt_unix.sleep 2.0;;
let b () = Lwt_unix.sleep 2.0;;
Lwt_main.run begin a () >>= b end;;
```

Examples: Sequential Composition(?)

```
let a = Lwt_unix.sleep 2.0;;
let b = Lwt_unix.sleep 2.0;;
Lwt_main.run begin a >>= fun () -> b end;;
```

Examples: Concurrent Composition

```
Lwt_main.run begin
  let a = Lwt_unix.sleep 2.0 in
  let b = Lwt_unix.sleep 2.0 in
  a >>= fun () -> b
end;;
```

Examples: Concurrent Composition

```
Lwt_main.run begin
  let a = Lwt_unix.sleep 2.0 in
  let b = Lwt_unix.sleep 2.0 in
  Lwt.choose [a; b]
end;;
```

More Examples

- `delay.ml` - try and predict the executions!

More Examples

- `delay.ml` - try and predict the executions!
- `crawl.ml` - interactive "game" loop

More Examples

- `delay.ml` - try and predict the executions!
- `crawl.ml` - interactive "game" loop
- `client.ml` and `server.ml` - working chat server!