# Linguagens e Ambientes de Programação (Aula Teórica 15)

**LEI - Licenciatura em Engenharia Informática**

**João Costa Seco (joao.seco@fct.unl.pt)**

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

# Agenda

- I/O assíncrono, Promessas e Futuros

# Concorrência

- Capacidade de realizar várias computações que se sobrepõem no tempo, e não exigir que sejam feitas em sequência.
  - Interfaces gráficos; para assegurar respostas rápida a acções do utilizador.
  - Folhas de cálculo; para recalcular todos os cálculos sem interrupções.
  - Web Browser; para carregar e mostrar páginas de forma incremental.
  - Servidores; para atender vários clientes sem os fazer esperar.
- Como?
  - Entrelaçamento: comutando entre as várias computações ativas rapidamente.
  - Paralelismo: utilizando vários processadores físicos (multi-core).

# Não determinismo

- Um programa que pode ter resultados diferentes de cada vez que é executado é um programa não determinista.
- A introdução de concorrência causa não determinismo, ao não fixar a ordem pela qual operações são executadas.
- Quando dois programas imperativos partilham variáveis de estado, as possibilidades de alteração desse mesmo estado são indeterminadas. Logo, não é fácil prever o comportamento exacto dos programas.
- As interações/interferências entre programas são benignas ou malignas.
- As linguagens funcionais tornam mais fácil o raciocínio sobre programas porque uma expressão pura denota sempre o mesmo valor.

# Promessas ou Futuros

- Representam computações que ainda não acabaram mas que irão denotar um valor algures num instante futuro (diferido).
- Normalmente associadas a uma computação concorrente ao thread principal.

```
async function getNames() {
  let response = await fetch('https://server/users')
  let users = await users.json()
  return users.map(user ⇒ user.name)
}
```

- Async (Jane Street) e Lwt (Ocsigen) são duas bibliotecas populares para implementar computação assíncrona em OCaml.

# Promessas ao estilo Lwt

- São abstrações de dados para um modelo de computação assíncrona.

- As promessas são referências, o seu valor pode mudar.
- Quando é criada não contém nada.
- Uma promessa pode ser cumprida e preenchida por um valor
- Uma promessa pode ser rejeitada (preenchida por uma exceção)
- Em ambos os casos diz-se resolvida.

# Assinatura do Módulo Promessa

```
(** A signature for Lwt-style promises, with better names *)
module type PROMISE = sig
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  val make : unit → 'a promise * 'a resolver

  (** [return x] is a new promise that is already fulfilled with value
      [ x ]. *)
  val return : 'a → 'a promise

  (** [state p] is the state of the promise *)
  val state : 'a promise → 'a state

  (** [fulfill r x] fulfills the promise [p] associated with [r] with
      value [ x ], meaning that [state p] will become [Fulfilled x].
      Requires: [p] is pending. *)
  val fulfill : 'a resolver → 'a → unit

  (** [reject r x] rejects the promise [p] associated with [r] with
      exception [ x ], meaning that [state p] will become [Rejected x].
      Requires: [p] is pending. *)
  val reject : 'a resolver → exn → unit
end
```

[1] ✓ 0.0s OCaml

# Implementação do Módulo Promessa

```
module Promise : PROMISE = struct
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise = 'a state ref

  type 'a resolver = 'a promise

  (** [write_once p s] changes the state of [p] to be [s]. If [p] and
      [s] are both pending, that has no effect. Raises: [Invalid_arg] if
      the state of [p] is not pending. *)
  let write_once p s =
    if !p = Pending then p := s else invalid_arg "cannot write twice"

  let make () =
    let p = ref Pending in
    (p, p)

  let return x = ref (Fulfilled x)

  let state p = !p

  let fulfill r x = write_once r (Fulfilled x)

  let reject r x = write_once r (Rejected x)
end
```

[4] ✓ 0.0s OCaml

```
module Promise : PROMISE
```

# I/O Síncrona

```
let file = "example.dat"
let message = "Hello!"

let () =
  let oc = open_out file in
  Printf.fprintf oc "%s\n" message;
  close_out oc;

  let ic = open_in file in
  try
    let line = input_line ic in
    print_endline line;
    flush stdout;
    close_in ic
  with e →
    close_in_noerr ic;
    raise e
```

[ ]

```
# ignore(input_line stdin); print_endline "done";;
<type your own input here>
done
- : unit = ()
```

https://ocaml.org/docs/file-manipulation

https://cs3110.github.io/textbook/chapters/ds/promises.html

# I/O Assíncrona

- As primitivas de I/O não bloqueiam à espera que termine a operação de I/O.
- O valor que denotam é uma promessa.
- O tipo da promessa é mascarado pelo `utop` mas é de facto `string Lwt.t`

```
#require "lwt.unix";;
open Lwt_io;;
let p = read_line stdin;;
Lwt.state p;;
```

[2] ✓ 0.0s

```
val p : string Lwt.t = <abstr>

- : string Lwt.state = Lwt.Sleep
```

A notação desta imagem é a notação nativa do módulo lwt

# Callbacks

- Uma função callback de uma promessa é uma função que é chamada aquando da realização da mesma.
- Liga-se uma função callback a uma promessa através da função `bind`.

```
[3]  let print_the_string str = Lwt_io.printf "The string is: %S\n" str;;

     Lwt.bind p print_the_string

...  val print_the_string : string → unit Lwt.t = <fun>
```

```
[3]  Lwt.bind
     ✓ 0.0s

...  - : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>
```

# Callbacks 2

- Várias operações assíncronas podem ser encadeadas, para manter o determinismo/sequencialidade.
- A função `bind` combina a função `run` espera pelo total resultado e devolve o valor final.

```
[4]
open Lwt_io

let p =
  Lwt.bind (read_line stdin) (fun s1 →
    Lwt.bind (read_line stdin) (fun s2 →
      Lwt_io.printf "%s\n" (s1^s2)))

let _ = Lwt_main.run p
```

```
[5]  ✓ 0.0s
Lwt.bind;;

Lwt_main.run
···  - : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>

···  - : 'a Lwt.t → 'a = <fun>
```

```
utop # #use "teoricas/LAP2024-15/promises.ml";;
val p : unit Lwt.t = <abstr>
one
two
- : unit = ()
onetwo
```

# Callbacks 2

- Várias operações assíncronas podem ser encadeadas, para manter o determinismo/sequencialidade.
- A função `bind` combina a função `run` espera pelo total resultado e devolve o valor final.

>>=

```
open Lwt_io
open Lwt.Infix

let p =
  read_line stdin >>= fun s1 ->
  read_line stdin >>= fun s2 ->
  Lwt_io.printf "%s\n" (s1^s2)

let _ = Lwt_main.run p
```

```
Lwt.bind;;

Lwt_main.run
```

[5] ✓ 0.0s

```
- : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>

- : 'a Lwt.t → 'a = <fun>
```

```
utop # #use "teoricas/LAP2024-15/promises.ml";;
val p : unit Lwt.t = <abstr>
one
two
- : unit = ()
onetwo
```

# Callbacks 2

- Várias operações assíncronas podem ser encadeadas, para manter o determinismo/sequencialidade.
- A função `bind` combina a função `run` espera pelo total resultado e devolve o valor final.

```
Lwt.bind;;

Lwt_main.run
```
[5] ✓ 0.0s
```
- : 'a Lwt.t → ('a → 'b Lwt.t) → 'b Lwt.t = <fun>

- : 'a Lwt.t → 'a = <fun>
```

>>=

```
open Lwt_io
open Lwt.Infix

let p =
  read_line stdin >>= fun s1 →
  read_line stdin >>= fun s2 →
  Lwt_io.printf "%s\n" (s1^s2)
```
[ ]

```
async function getNames() {
  return fetch('https://jsonplaceholder.typicode.com/users')
                .then( response ⇒ response.json() )
                .then( data ⇒ data.map(user ⇒ user.name));
}


getNames().then( names ⇒ console.log(names) );
```
JavaScript

```
utop
val
one
two
- : unit = ()
onetwo
```

# Callbacks implemented

https://cs3110.github.io/textbook/chapters/ds/promises.html

```
(** A signature for Lwt-style promises, with better names *)
module type PROMISE = sig
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  unit -> 'a promise * 'a resolver
  val make : unit → 'a promise * 'a resolver

  (** [return x] is a new promise that is already fulfilled with value [ x ]. *)
  'a -> 'a promise
  val return : 'a → 'a promise

  (** [state p] is the state of the promise *)
  'a promise -> 'a state
  val state : 'a promise → 'a state

  (** [fulfill r x] resolves the promise [p] associated with [r] with
      value [ x ], meaning that [state p] will become [Fulfilled x].
      Requires: [p] is pending. *)
  'a resolver -> 'a -> unit
  val fulfill : 'a resolver → 'a → unit

  (** [reject r x] rejects the promise [p] associated with [r] with
      exception [ x ], meaning that [state p] will become [Rejected x].
      Requires: [p] is pending. *)
  'a resolver -> exn -> unit
  val reject : 'a resolver → exn → unit

  (** [p >>= c] registers callback [c] with promise [p].
      When the promise is fulfilled, the callback will be run
      on the promises's contents.  If the promise is never
      fulfilled, the callback will never run. *)
  'a promise -> ('a -> 'b promise) -> 'b promise
  val ( >>= ) : 'a promise → ('a → 'b promise) → 'b promise
end
```

# Callbacks implemented

```
module Promise : PROMISE = struct
  type 'a state = Pending | Fulfilled of 'a | Rejected of exn

  (** RI (representation invariant): the input may not be [Pending] *)
  type 'a handler = 'a state → unit

  (** RI: if [state <> Pending] then [handlers = []]. *)
  type 'a promise = {
    mutable state : 'a state;
    mutable handlers : 'a handler list
  }
```

# Callbacks implemented

```
('a state -> unit) -> 'a promise -> unit
let enqueue
      (handler : 'a state → unit)
      (promise : 'a promise) : unit
    =
    promise.handlers ← handler :: promise.handlers

type 'a resolver = 'a promise

(** [write_once p s] changes the state of [p] to be [s].  If [p] and [s]
      are both pending, that has no effect.
      Raises: [Invalid_arg] if the state of [p] is not pending. *)
'a resolver -> 'a state -> unit
let write_once p s =
  if p.state = Pending
  then p.state ← s
  else invalid_arg "cannot write twice"
```

# Callbacks implemented

```
unit -> 'a resolver * 'a resolver
let make () =
  let p = {state = Pending; handlers = []} in
  p, p

'a -> 'a resolver
let return x =
  {state = Fulfilled x; handlers = []}

'a resolver -> 'a state
let state p = p.state

(** requires: [st] may not be [Pending] *)
'a resolver -> 'a state -> unit
let resolve (r : 'a resolver) (st : 'a state) =
  assert (st <> Pending);
  let handlers = r.handlers in
  r.handlers ← [];
  write_once r st;
  List.iter (fun f → f st) handlers
```

# Callbacks implemented

```
'a resolver -> exn -> unit
let reject r x =
  resolve r (Rejected x)

'a resolver -> 'a -> unit
let fulfill r x =
  resolve r (Fulfilled x)

'a resolver -> 'a handler
let handler (resolver : 'a resolver) : 'a handler
  = function
    | Pending → failwith "handler RI violated"
    | Rejected exc → reject resolver exc
    | Fulfilled x → fulfill resolver x
```

# Callbacks implemented

```
('a -> 'b resolver) -> 'b resolver -> 'a handler
let handler_of_callback
    (callback : 'a → 'b promise)
    (resolver : 'b resolver) : 'a handler
  = function
    | Pending → failwith "handler RI violated"
    | Rejected exc → reject resolver exc
    | Fulfilled x →
      let promise = callback x in
      match promise.state with
      | Fulfilled y → fulfill resolver y
      | Rejected exc → reject resolver exc
      | Pending → enqueue (handler resolver) promise

'a resolver -> ('a -> 'b resolver) -> 'b resolver
let ( >>= )
    (input_promise : 'a promise)
    (callback : 'a → 'b promise) : 'b promise
  =
  match input_promise.state with
  | Fulfilled x → callback x
  | Rejected exc → {state = Rejected exc; handlers = []}
  | Pending →
    let output_promise, output_resolver = make () in
    enqueue (handler_of_callback callback output_resolver) input_promise;
    output_promise
end
```