ocsigen

Programmation asynchrone avec LWT

Pierre Chambart

Journées Francophones de Leçons sur Ocsigen

Lancez Emacs, lancez un toplevel

et tapotez:

```
1 : #directory "+../toplevel";;  (* pour les opam users *)
2 : #use "topfind";;             (* charge ocamlfind *)
3 : #camlp4o;;                   (* utilise camlp4 *)
4 : #require "lwt.syntax";;      (* charge la syntaxe lwt *)
5 : #require "lwt.unix";;        (* charge lwt et lwt.unix *)
6 : open Lwt
```

Pour pouvoir relancer le toplevel, mettez ça dans le fichier ~/.ocamlinit

La base

Les 2 fonctions à la base de LWT

```
1 : return ;;
2 : - : 'a -> 'a Lwt.t = <fun>
3 : bind ;;
4 : - : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t = <fun>
```

et celle pratique pour apprendre:[a]

```
1 : state ;;
2 : - : 'a Lwt.t -> 'a Lwt.state = <fun>
```

[a] mais qui ne s'utilise pas en vrai

Premiers threads

```
1 : let t = return 1
2 : val t : int Lwt.t = <abstr>
3 : state t
4 : - : int Lwt.state = Return 1
5 : let t2 = bind t (fun x -> return (x+1))
6 : val t2 : int Lwt.t = <abstr>
7 : state t2
8 : - : int Lwt.state = Return 2
```

Un peu de syntaxe

```
1 : let (>>=) = Lwt.bind (* déjà défini dans Lwt *)
2 : let t2 = bind t (fun x -> return (x+1))
3 : let t2 = t >>= fun x -> return (x+1)
4 : let t2 =
5 :   lwt x = t in
6 :   return (x+1)
7 :
8 : let t3 = return () >>= fun _ -> return 1
9 : let t3 = return () >> return 1
```

Qu'importe la précédence

t3 est parenthésé comme t4, mais a exactement la même sémantique que t5 [a]

```
1 : let t3 = t
2 :          >>= fun x -> return (x+1)
3 :          >>= fun y -> return (float y)
4 :
5 : let t4 =
6 :   t >>= fun x ->
7 :   begin return (x+1) >>= fun y -> return (float y) end
8 :
9 : let t5 =
10 :   begin t >>= fun x -> return (x+1) end
11 :   >>= fun y -> return (float y)
```

[a] du coup pas besoin de se poser de questions

Les threads endormis

```
1 : wait ;;
2 : - : unit -> 'a Lwt.t * 'a Lwt.u = <fun>
3 : wakeup ;;
4 : - : 'a Lwt.u -> 'a -> unit = <fun>
```

Les threads endormis

```
1 : let thread_endormi, reveilleur = wait ();;
2 : val thread_endormi : '_a Lwt.t = <abstr>
3 : val reveilleur : '_a Lwt.u = <abstr>
4 :
5 : state thread_endormi;;
6 : - : '_a Lwt.state = Sleep
7 : wakeup reveilleur 42;;
8 : - : unit = ()
9 : state thread_endormi;;
10 : - : int Lwt.state = Return 42
```

Les threads endormis

```
1 : let thread_endormi, reveilleur = wait ();;
2 : val thread_endormi : '_a Lwt.t = <abstr>
3 : val reveilleur : '_a Lwt.u = <abstr>
4 :
5 : state thread_endormi;;
6 : - : '_a Lwt.state = Sleep
7 : wakeup reveilleur 42;;
8 : - : unit = ()
9 : state thread_endormi;;
10 : - : int Lwt.state = Return 42
```

Enfin des choses interessantes

L'état de réveil se propage à tous les descendants:

```
1 : let t,w = wait ()
2 : let t2 = t >>= fun x -> return (x+1)
3 : state t2;;
4 : - : int Lwt.state = Sleep
5 : wakeup w 2;;
6 : state t2;;
7 : - : int Lwt.state = Return 3
```

Un thread ne peut être réveillé qu'une fois:

```
1 : wakeup w 42;;
2 : Exception: Invalid_argument "Lwt.wakeup_result".
```

Wait caché

```
1 : let t = Lwt_unix.sleep 1.;;
2 : val t : unit Lwt.t = <abstr>
3 : state t;;
4 : - : unit Lwt.state = Sleep
5 : Lwt_main.run t;;
6 : - : unit = ()
7 : state t;;
8 : - : unit Lwt.state = Return ()
```

Tous les appels systèmes font des wait cachés et sont réveillé par le scheduler.

Lwt_main.run t continue d'exécuter tous les threads unix tant que le thread t n'est pas terminé.

Wait caché

```
1 : let t =
2 :   print_endline "prepare";
3 :   lwt () = Lwt_unix.sleep 0.1 in
4 :   print_endline "start";
5 :   lwt () = Lwt_unix.sleep 1. in
6 :   print_endline "stop";
7 :   return ()
8 :
9 : prepare
10 : - : unit Lwt.t
11 :
12 : let () = Lwt_main.run (Lwt_unix.sleep 2.)
13 :
14 : start
15 : stop
16 : - : unit
```

Exercice 0

Boucle qui print toutes les secondes

```
1 : let t = loop 3
2 : val t : unit Lwt.t = <fun>
3 :
4 : let () = Lwt_main.run t
5 : 3 (* attente d'une secondes entre chaque print *)
6 : 2
7 : 1
8 : 0
```

Solution 0

```
1 : let rec print_loop i =
2 :   if i < 0
3 :   then return ()
4 :   else
5 :     Lwt_unix.sleep 1. >>=
6 :     fun () ->
7 :       Printf.printf "%i\n%!" i;
8 :       print_loop (i-1)
```

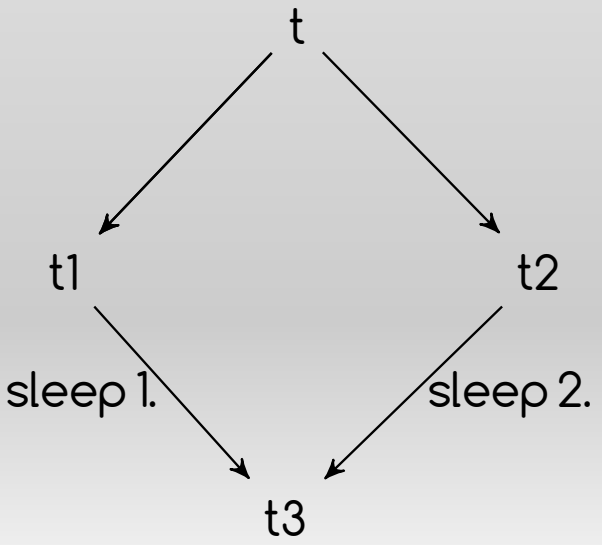
pas de problème de tail récursion

```
1 : let rec fact_tail acc i =
2 :   if i = 1
3 :   then return acc
4 :   else return (i-1) >>= fact_tail (acc*i)
5 : let fact = fact_acc 1
6 : let v = fact 10000000
7 : state v;;
8 : - : int Lwt.state = Return 0
```

threads en parallele

```
1 : let t = pause ()
2 : let t1 = t >>= fun () -> Lwt_unix.sleep 1.
3 :            >>= fun () -> Lwt_io.printf "t1\n"
4 : let t2 = t >>= fun () -> Lwt_unix.sleep 2.
5 :            >>= fun () -> Lwt_io.printf "t2\n"
6 : let t3 = join [t1; t2]
7 : Lwt_main.run t3;;
```

```
1 : t1
2 : t2
3 : - : unit = ()
```



Coopération

```
1 : let t = pause ()
2 : let t1 = t >>= fun () -> Lwt_unix.sleep 1.
3 :            >>= fun () -> Lwt_io.printf "t1\n"
4 : let t2 = t >>= fun () -> Unix.sleep 2;
5 :                          print_endline "ici!";
6 :                          return ()
7 :            >>= fun () -> Lwt_io.printf "t2\n"
8 : let t3 = join [t1; t2]
9 : Lwt_main.run t3;;
```

```
1 : ici
2 : t2
3 : t1
4 : - : unit = ()
```

Exercice 1

Écrire join en utilisant bind

```
1 : join
2 : - : unit Lwt.t list -> unit Lwt.t = <fun>
```

Join termine quand l'ensemble de ses paramètres terminent.

Solution 1

```
1 : let rec join = function
2 :   | [] -> return ()
3 :   | t::q -> t >>= fun () -> join q
```

Exercice 2

Écrire `map_serial`

```
1 : map_serial
2 : - : ('a -> unit Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

`map_serial f l` exécute `f` sur tous les éléments de `l` dans l'ordre de la liste.

```
1 : let f (t,s) = Lwt_unix.sleep t >> Lwt_io.print s >> return s
2 : let t = map_serial f [0.2,"1\n";
3 :                       0.1,"2\n";
4 :                       0.3,"3\n"]
5 : let l = Lwt_main.run t
6 :
7 : 1
8 : 2
9 : 3
10 : val l : string list = ["1\n"; "2\n"; "3\n"]
```

Solution 2

```
1 : let rec map_serial f = function
2 :   | [] -> return []
3 :   | t::q ->
4 :     lwt t' = f t in
5 :     lwt q' = map_serial f q in
6 :     return (t'::q')
```

Exercice 3

Écrire `map_parallel`

```
1 : map_parallel
2 : - : ('a -> unit Lwt.t) -> 'a list -> 'b list Lwt.t = <fun>
```

`map_parallel f l` exécute `f` sur tous les éléments de `l` en parallèle.

```
1 : let f (t,s) = Lwt_unix.sleep t >> Lwt_io.print s >> return s
2 : let t = map_serial f [0.2,"1\n";
3 :                       0.1,"2\n";
4 :                       0.3,"3\n"]
5 : let l = Lwt_main.run t
6 :
7 : 2
8 : 1
9 : 3
10 : val l : string list = ["1\n"; "2\n"; "3\n"]
```

Solution 3

```
1: let rec map_parallele f = function
2:   | [] -> return []
3:   | t::q ->
4:     let t1 = f t in
5:     lwt t2 = map_p f q in
6:     lwt t1 = t1 in
7:     return (t1 :: t2)
```

Exemple: requettes http

```
1 : #thread;;
2 : #require "ocsigenserver";;
3 : Ocsigen_http_client.get_url;;
4 : - : ?headers:Http_headers.t -> string ->
5 :       Ocsigen_http_frame.t Lwt.t = <fun>
6 :
7 : let frame_to_string = function
8 :   | { Ocsigen_http_frame.frame_content = Some v } ->
9 :       Ocsigen_stream.string_of_stream 100000
10 :         (Ocsigen_stream.get v)
11 :   | _ -> return ""
12 : val frame_to_string : Ocsigen_http_frame.t -> string Lwt.t = <fun
```

Exemple: requettes http

```
1: let t = map_parallele Ocsigen_http_client.get_url
2:   [ "http://ocsigen.org/";
3:     "http://www.irill.org/";
4:     "http://www.ocamlpro.com/" ]
5:
6: let t2 = t >>= map_parallele frame_to_string
7: let v = Lwt_main.run t2
8:
9: ... plein de html ...
```

Exceptions et LWT

```
1 : let t = Ocsigen_http_client.get_url "http://oscygen.org/";;
2 : Lwt_main.run t;;
3 : Exception: Ocsigen_lib.Ip_address.No_such_host.
4 : state t;;
5 : - : Ocsigen_http_frame.t Lwt.state =
6 :       Fail Ocsigen_lib.Ip_address.No_such_host
```

Exceptions et LWT

```
1 : fail;;
2 : - : exn -> 'a Lwt.t = <fun>
3 : let t = fail Not_found
4 : state t
5 : - : 'a Lwt.state = Fail Not_found
6 : let t2 = t >>= fun x -> return (x + 1)
7 : state t2
8 : - : 'a Lwt.state = Fail Not_found
```

Rattraper des exceptions

```
1 : catch;;
2 : - : (unit -> 'a Lwt.t) -> (exn -> 'a Lwt.t) -> 'a Lwt.t = <fun>
3 : let t =
4 :   catch (fun () -> fail Not_found)
5 :         (function
6 :             Not_found -> return "not_found"
7 :           | e -> fail e)
8 : state t;;
9 : - : string Lwt.state = Return "not_found"
```

La bonne manière des gerer les exceptions

```
1 : open Lwt
2 : let t =
3 :   try_lwt
4 :     raise_lwt (Failure "truc")
5 :   with Not_found -> return "chose"
6 : let _ = Lwt_main.run t
```

`raise_lwt` et `try_lwt` avec `-lwt-debug` génèrent du code qui propage les backtrace.

```
1 : ocamlfind ocamlc -g -syntax camlp4o
2 :   -package lwt.syntax -ppopt -lwt-debug
3 :   -package lwt.unix -linkpkg -o exc exception.ml
4 : ./exc
5 : Fatal error: exception Failure("truc")
6 : Raised at file "exception.ml", line 6, characters 15-29
7 : Re-raised at file "exception.ml", line 5, characters 2-75
8 : Re-raised at file "src/core/lwt.ml", line 782, characters 22-23
9 : Called from file "src/unix/lwt_main.ml", line 34, characters 8-18
10 : Called from file "exception.ml", line 9, characters 8-22
```

Interrompre un thread

```
1 : let t,w = task () (* presque comme wait *)
2 : val t : '_a Lwt.t = <abstr>
3 : val w : '_a Lwt.u = <abstr>
4 : state t;;
5 : - : '_a Lwt.state = Sleep
6 : cancel t;;
7 : - : unit = ()
8 : state t;;
9 : - : '_a Lwt.state = Fail Lwt.Canceled
```

Interrompre un thread

```
1 : let t,w = task ()
2 : on_cancel t (fun () -> print_endline "cancel");;
3 : let t2 = t >>= return
4 : cancel t2
5 :
6 : cancel
```

Timeout

```
1: let timeout d = sleep d >> Lwt.fail Timeout
2:
3: let with_timeout d f = Lwt.pick [timeout d; Lwt.apply f ()]
4: (* Lwt_unix.with_timeout *)
```

Tour de LWT: core

- `Lwt_stream`: flux de threads
- `Lwt_react`: integration avec React
- `Lwt_pool`: gestion de resources limitées
- `Lwt_list`: map, fold, iter, etc
- `Lwt_mutex`: parfois il en faut
- `Lwt_condition`: attente d'un condition

Tour de LWT: unix

- `Lwt_unix`: `Unix` pour LWT
- `Lwt_io`: lecture/écriture de fichier/socket
- `Lwt_log`: système de log complet
- `Lwt_throttle`: limitation de débit
- `Lwt_process`: gestion de processus
- `Lwt_bytes`: lecture/écriture optimisées
- `Lwt_daemon`: processus lancé en daemon

Tour de LWT: autres

- `Lwt_ssl`: sockets SSL/TLS
- `Lwt_glib`: integration dans la boucle glib
- `Lwt_preemptive`: interraction avec les threads préemptifs
- `lambda-term`: gestion du terminal
- la votre que j'ai oublié ?

interraction avec les threads préemptifs

```
1 : let read_lines f =
2 :   try while true do
3 :         Unix.sleep 1;
4 :         Printf.printf "ligne: %s\n%!" (f ())
5 :   done with
6 :     End_of_file ->
7 :     Printf.printf "fini\n%!"
8 :
9 : let read_lines_lwt f =
10 :   let f' () = Lwt_preemptive.run_in_main f in
11 :   read_lines f'
12 :
13 : let t =
14 :   Lwt_preemptive.detach
15 :     read_lines_lwt (fun () -> Lwt_io.read_line Lwt_io.stdin)
16 :
17 : let () = Lwt_main.run t
```

gtk et LWT

```
1 : #use "topfind";;
2 : #require "lwt.simple-top";;
3 : #require "lwt.glib";;
4 : #require "lablgtk2";;
5 :
6 : GMain.init ();;
7 : Lwt_glib.install ();;
8 :
9 : let w = GWindow.window ();;
10 : w#show ();;
```

stream

```
1 : open Lwt
2 : let count_lines channel =
3 :   let line_stream =
4 :     Lwt_stream.from (fun () -> Lwt_io.read_line_opt channel) in
5 :   Lwt_stream.fold (fun _ n -> n + 1) line_stream 0
6 :
7 : let f file =
8 :   try_lwt
9 :     Lwt_io.with_file Lwt_io.input file count_lines
10 :    >>= Lwt_io.printf "%s: %i\n" file
11 :  with exn -> Lwt_log.error ~exn "erreur !"
12 :
13 : let stdin_stream =
14 :   Lwt_stream.from (fun () -> Lwt_io.read_line_opt Lwt_io.stdin)
15 :
16 : let () = Lwt_main.run (Lwt_stream.iter_p f stdin_stream)
```

OCamlPro

- Service autour de OCaml
- Cours
- Convaincre votre boss de vous autoriser à faire du OCaml
- Dévellopement/maintenance

Enfin tout ce qui à trait à OCaml...

ocsigen

Et maintenant à Eliom

Journées Francophones de Leçons sur Ocsigen

Pour les curieux, ce qui est généré avec -lwt-debug

```
1 : open Lwt
2 : let t =
3 :   try_lwt
4 :     raise_lwt (Failure "truc")
5 :   with Not_found -> return "chose"
6 : let _ = Lwt_main.run t
```

```
1 : open Lwt
2 : Lwt.backtrace_catch
3 :   (fun exn -> try raise exn with exn -> exn)
4 :   (fun () -> Lwt.fail
5 :     (try raise (Failure "truc") with exn -> exn))
6 :   (function
7 :    | Not_found -> return "chose"
8 :    | exn -> Lwt.fail exn)
```

Fichier et ligne de l'erreur avec Lwt_log

```
1: let () =
2:   Lwt_log.default :=
3:     Lwt_log.channel
4:       ~template:"$(date): $(section), ($(loc-file),$(loc-line )): 
5:       ~close_mode:`Keep
6:       ~channel:Lwt_io.stdout ()
```