Lwt user manual

Jérémie Dimino

December 27, 2012

Contents

1 Introduction

When writing a program, a common developer's task is to handle IO operations. Indeed most software interact with several different resources, such as:

- the kernel, by doing system calls
- the user, by reading the keyboard, the mouse, or any input device
- a graphical server, to build graphical user interface
- other computers, by using the network
- ...

When this list contains only one item, it is pretty easy to handle. However as this list grows it becomes harder and harder to make everything works together. Several choices have been proposed to solve this problem:

- using a main loop, and integrate all components we are interacting with into this main loop.
- using preemptive system threads

Both solutions have their advantages and their drawbacks. For the first one, it may work, but it becomes very complicated to write a piece of asynchronous sequential code. The typical example is graphical user interfaces freezing and not redrawing themselves because they are waiting for some blocking part of the code to complete.

If you already wrote code using preemptive threads, you should know that doing it right with threads is a hard job. Moreover system threads consume non negligible resources, and so you can only launch a limited number of threads at the same time. Thus this is not a real solution.

`Lwt` offers a new alternative. It provides very light-weight cooperative threads; "launching" a thread is a very fast operation, it does not require a new stack, a new process, or anything else. Moreover context switches are very fast. In fact, it is so easy that we will launch a thread for every system call. And composing cooperative threads will allow us to write highly asynchronous programs.

In a first part, we will explain the concepts of `Lwt`, then we will describe the many sub-libraries of `Lwt`.

2 The Lwt core library

In this section we describe the basics of `Lwt`. It is advised to start an ocaml toplevel and try the given code examples. To start, launch `ocaml` in a terminal or in emacs with the tuareg mode, and type:

```
# #use "topfind";;
# #require "lwt.simple-top";;
```

`lwt.simple-top` makes sure `Lwt` threads can run while using the toplevel. You do not need it if your are using `utop`.

2.1 Lwt concepts

Let's take a classical function of the `Pervasives` module:

```
# Pervasives.input_char;
- : in_channel -> char = <fun>
```

This function will wait for a character to come on the given input channel, and then return it. The problem with this function is that it is blocking: while it is being executed, the whole program will be blocked, and other events will not be handled until it returns.

Now let's look at the lwt equivalent:

```
# Lwt_io.read_char;;
- : Lwt_io.input_channel -> char Lwt.t = <fun>
```

As you can see, it does not return a character but something of type `char Lwt.t`. The type `'a Lwt.t` is the type of threads returning a value of type `'a`. Actually the `Lwt_io.read_char` will try to read a character from the given input channel and *immediatly* returns a light-weight thread.

Now, let's see what we can do with a `Lwt` thread. The following code creates a pipe, and launches a thread reading on the input side:

```
# let ic, oc = Lwt_io.pipe ();;
val ic : Lwt_io.input_channel = <abstr>
val oc : Lwt_io.output_channel = <abstr>
# let t = Lwt_io.read_char ic;;
val t : char Lwt.t = <abstr>
```

We can now look at the state of our newly created thread:

```
# Lwt.state t;;
- : char Lwt.state = Sleep
```

A thread may be in one of the following states:

- `Return x`, which means that the thread has terminated successfully and returned the value `x`
- `Fail exn`, which means that the thread has terminated, but instead of returning a value, it failed with the exception `exn`
- `Sleep`, which means that the thread is currently sleeping and has not yet returned a value or an exception

The thread `t` is sleeping because there is currently nothing to read from the pipe. Let's write something:

```
# Lwt_io.write_char oc 'a';;
- : unit Lwt.t = <abstr>
# Lwt.state t;;
- : char Lwt.state = Return 'a'
```

So, after we write something, the reading thread has been awoken and has returned the value `'a'`.

2.2 Primitives for thread creation

There are several primitives for creating `Lwt` threads. These functions are located in the module `Lwt`.

Here are the main primitives:

- `Lwt.return : 'a -> 'a Lwt.t`
  creates a thread which has already terminated and returned a value
- `Lwt.fail : exn -> 'a Lwt.t`
  creates a thread which has already terminated and failed with an exception

- `Lwt.wait : unit -> 'a Lwt.t * 'a Lwt.u`
  creates a sleeping thread and returns this thread plus a wakener (of type `'a Lwt.u`) which must be used to wakeup the sleeping thread.

To wake up a sleeping thread, you must use one of the following functions:

- `Lwt.wakeup : 'a Lwt.u -> 'a -> unit`
  wakes up the thread with a value.

- `Lwt.wakeup_exn : 'a Lwt.u -> exn -> unit`
  wakes up the thread with an exception.

Note that this is an error to wakeup the same threads twice. `Lwt` will raise `Invalid_argument` if you try to do so.

With these informations, try to guess the result of each of the following expression:

```
# Lwt.state (Lwt.return 42);;
# Lwt.state (fail Exit);;
# let waiter, wakener = Lwt.wait ();;
# Lwt.state waiter;;
# Lwt.wakeup wakener 42;;
# Lwt.state waiter;;
# let waiter, wakener = Lwt.wait ();;
# Lwt.state waiter;;
# Lwt.wakeup_exn wakener Exit;;
# Lwt.state waiter;;
```

2.2.1 Primitives for thread composition

The most important operation you need to know is `bind`:

```
val bind : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

`bind t f` creates a thread which waits for `t` to terminate, then passes the result to `f`. If `t` is a sleeping thread, then `bind t f` will be a sleeping thread too, until `t` terminates. If `t` fails, then the resulting thread will fail with the same exception. For example, consider the following expression:

```
Lwt.bind
  (Lwt_io.read_line Lwt_io.stdin)
  (fun str -> Lwt_io.printlf "You typed %S" str)
```

This code will first wait for the user to enter a line of text, then print a message on the standard output.

Similarly to `bind`, there is a function to handle the case when `t` fails:

```
val catch : (unit -> 'a Lwt.t) -> (exn -> 'a Lwt.t) -> 'a Lwt.t
```

`catch f g` will call `f ()`, then waits for its termination, and if it fails with an exception `exn`, calls `g exn` to handle it. Note that both exceptions raised with `Pervasives.raise` and `Lwt.fail` are caught by `catch`.

2.2.2 Cancelable threads

In some case, we may want to cancel a thread. For example, because it has not terminated after a timeout. This can be done with cancelable threads. To create a cancelable thread, you must use the `Lwt.task` function:

```
val task : unit -> 'a Lwt.t * 'a Lwt.u
```

It has the same semantics as `Lwt.wait` except that the sleeping thread can be canceled with `Lwt.cancel`:

```
val cancel : 'a Lwt.t -> unit
```

The thread will then fail with the exception `Lwt.Canceled`. To execute a function when the thread is canceled, you must use `Lwt.on_cancel`:

```
val on_cancel : 'a Lwt.t -> (unit -> unit) -> unit
```

Note that it is also possible to cancel a thread which has not been created with `Lwt.task`. In this case, the deepest cancelable thread connected with the given thread will be cancelled.

For example, consider the following code:

```
# let waiter, wakener = Lwt.task ();;
val waiter : '_a Lwt.t = <abstr>
val wakener : '_a Lwt.u = <abstr>
# let t = bind waiter (fun x -> return (x + 1));;
val t : int Lwt.t = <abstr>
```

Here, cancelling `t` will in fact cancel `waiter`. `t` will then fail with the exception `Lwt.Canceled`:

```
# Lwt.cancel t;;
- : unit = ()
# Lwt.state waiter;;
- : int Lwt.state = Fail Lwt.Canceled
# Lwt.state t;;
- : int Lwt.state = Fail Lwt.Canceled
```

By the way, it is possible to prevent a thread from being canceled by using the function `Lwt.protected`:

```
val protected : 'a Lwt.t -> 'a Lwt.t
```

Canceling `(proctected t)` will have no effect on `t`.

2.2.3 Primitives for multi-thread composition

We now show how to compose several concurrent threads. The main functions for this are in the `Lwt` module: `join`, `choose` and `pick`.

The first one, `join` takes a list of threads and waits for all of them to terminate:

```
val join : unit Lwt.t list -> unit Lwt.t
```

Moreover, if at least one thread fails, `join l` will fail with the same exception as the first to fail, after all threads terminate.

Similarly `choose` waits for at least one thread to terminate, then returns the same value or exception:

```
val choose : 'a Lwt.t list -> 'a Lwt.t
```

For example:

```
# let waiter1, wakener1 = Lwt.wait ();;
val waiter1 : '_a Lwt.t = <abstr>
val wakener1 : '_a Lwt.u = <abstr>
# let waiter2, wakener2 = Lwt.wait ();;
val waiter2 : '_a Lwt.t = <abstr>
val wakener : '_a Lwt.u = <abstr>
# let t = Lwt.choose [waiter1; waiter2];;
val t : '_a Lwt.t = <abstr>
# Lwt.state t;;
- : '_a Lwt.state = Sleep
# Lwt.wakeup wakener2 42;;
- : unit = ()
# Lwt.state t;;
- : int Lwt.state = Return 42
```

The last one, `pick`, is the same as `join` except that it cancels all other threads when one terminates.

2.2.4 Threads local storage

Lwt can store variables with different values on different threads. This is called threads local storage. For example, this can be used to store contexts or thread identifiers. The contents of a variable can be read with:

```
val Lwt.get : 'a Lwt.key -> 'a option
```

which takes a key to identify the variable we want to read and returns either `None` if the variable is not set, or `Some x` if it is. The value returned is the value of the variable in the current thread.

New keys can be created with:

```
val Lwt.new_key : unit -> 'a Lwt.key
```

To set a variable, you must use:

```
val Lwt.with_value : 'a Lwt.key -> 'a option -> (unit -> 'b) -> 'b
```

`with_value key value f` will execute `f` with the binding `key -> value`. The old value associated to `key` is restored after `f` terminates.

For example, you can use local storage to store thread identifiers and use them in logs:

```
let id_key = Lwt.new_key ()

let log msg =
  let thread_id =
    match Lwt.get id_key with
      | Some id -> id
      | None -> "main"
  in
  Lwt_io.printlf "%s: %s" thread_id msg

lwt () =
  Lwt.join [
    Lwt.with_value id_key (Some "thread 1") (fun () -> log "foo");
    Lwt.with_value id_key (Some "thread 2") (fun () -> log "bar");
  ]
```

2.2.5 Rules

`Lwt` will always try to execute as much as possible before yielding and switching to another cooperative thread. In order to make it work well, you must follow the following rules:

- do not write function that may takes time to complete without using `Lwt`,
- do not do IOs that may block, otherwise the whole program will hang. You must instead use asynchronous IOs operations.

2.3 The syntax extension

`Lwt` offers a syntax extension which increases code readability and makes coding using `Lwt` easier. To use it add the "lwt.syntax" package when compiling:

```
$ ocamlfind ocamlc -syntax camlp4o -package lwt.syntax -linkpkg -o foo foo.ml
```

Or in the toplevel (after loading topfind):

```
# #camlp4o;;
# #require "lwt.syntax";;
```

The following constructions are added to the language:

- `lwt` *pattern*$_1$ `=` *expr*$_1$ [`and` *pattern*$_2$ `=` *expr*$_2$ ...] `in` *expr*
  which is a parallel let-binding construction. For example in the following code:

```
lwt x = f () and y = g () in
expr
```

the thread `f ()` and `g ()` are launched concurrently and their results are then bound to `x` and `y` in the expression *expr*.

Of course you can also launch the two threads sequentially by writing your code like that:

```
lwt x = f () in
lwt y = g () in
expr
```

- `try_lwt` *expr* [`with` *pattern*$_1$ `->` *expr*$_1$ ...] [`finally` *expr'*]
  which is the equivalent of the standard `try-with` construction but for `Lwt`. Both exceptions raised by `Pervasives.raise` and `Lwt.fail` are caught.";
- `for_lwt` *ident* `=` *expr*$_{init}$ (`to` | `downto`) *expr*$_{final}$ `do` *expr* `done`
  which is the equivalent of the standard `for` construction but for `Lwt`.
- `raise_lwt` *exn*
  which is the same as `Lwt.fail` *exn* but with backtrace support.

2.3.1 Correspondence table

You might appreciate the following table to write code using lwt:

without `Lwt`	with `Lwt`
`let` $pattern_1$ `=` $expr_1$ `and` $pattern_2$ `=` $expr_2$ ... `and` $pattern_n$ `=` $expr_n$ `in` *expr*	`lwt` $pattern_1$ `=` $expr_1$ `and` $pattern_2$ `=` $expr_2$ ... `and` $pattern_n$ `=` $expr_n$ `in` *expr*

`try`
expr
`with`
`|` $pattern_1$ `->` $expr_1$
`|` $pattern_2$ `->` $expr_2$
...
`|` $pattern_n$ `->` $expr_n$
	`try_lwt`
	expr
	`with`
	`|` $pattern_1$ `->` $expr_1$
	`|` $pattern_2$ `->` $expr_2$
	...
	`|` $pattern_n$ `->` $expr_n$

`for` *ident* `=` $expr_{init}$ `to` $expr_{final}$ `do`
expr
`done`
	`for_lwt` *ident* `=` $expr_{init}$ `to` $expr_{final}$ `do`
	expr
	`done`

`raise` *exn*	`raise_lwt` *exn*
`assert` *expr*	`assert_lwt` *expr*
`match` *expr* `with` `|` $pattern_1$ `->` $expr_1$ `|` $pattern_2$ `->` $expr_2$ ... `|` $pattern_n$ `->` $expr_n$	`match_lwt` *expr* `with` `|` $pattern_1$ `->` $expr_1$ `|` $pattern_2$ `->` $expr_2$ ... `|` $pattern_n$ `->` $expr_n$

`while` *expr* `do`
expr
`done`
	`while_lwt` *expr* `do`
	expr
	`done`

2.4 Backtrace support

When using `Lwt`, exceptions are not recorded by the ocaml runtime, and so you don't get backtraces. However it is possible to get them when using the syntax extension. All you have to do is to pass the `-lwt-debug` switch to `camlp4`:

```
$ ocamlfind ocamlc -syntax camlp4o -package lwt.syntax \
    -ppopt -lwt-debug -linkpkg -o foo foo.ml
```

2.5 Other modules of the core library

The core library contains several modules that only depend on `Lwt`. The following naming convention is used in `Lwt`: when a function takes as argument a function returning a thread that is going to be executed sequentially, it is suffixed with "`_s`". And when it is going to be executed concurrently, it is suffixed with "`_p`". For example, in the `Lwt_list` module we have:

```
val map_s : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t
val map_p : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t
```

2.5.1 Mutexes

`Lwt_mutex` provides mutexes for `Lwt`. Its use is almost the same as the `Mutex` module of the thread library shipped with OCaml. In general, programs using `Lwt` do not need a lot of mutexes. They are only usefull for serialising operations.

2.5.2 Lists

The `Lwt_list` module defines iteration and scanning functions over lists, similar to the ones of the `List` module, but using functions that return a thread. For example:

```
val iter_s : ('a -> unit Lwt.t) -> 'a list -> unit Lwt.t
val iter_p : ('a -> unit Lwt.t) -> 'a list -> unit Lwt.t
```

In `iter_s f l`, `iter_s` will call f on each elements of `l`, waiting for completion between each element. On the contrary, in `iter_p f l`, `iter_p` will call f on all elements of `l`, then wait for all the threads to terminate.

2.5.3 Data streams

`Lwt` streams are used in a lot of places in `Lwt` and its sub libraries. They offer a high-level interface to manipulate data flows.

A stream is an object which returns elements sequentially and lazily. Lazily means that the source of the stream is touched only for new elements when needed. This module contains a lot of stream transformation, iteration, and scanning functions.

The common way of creating a stream is by using `Lwt_stream.from` or by using `Lwt_stream.create`:

```
val from : (unit -> 'a option Lwt.t) -> 'a Lwt_stream.t
val create : unit -> 'a Lwt_stream.t * ('a option -> unit)
```

As for streams of the standard library, `from` takes as argument a function which is used to create new elements.

`create` returns a function used to push new elements into the stream and the stream which will receive them.

For example:

```
# let stream, push = Lwt_stream.create ();;
val stream : '_a Lwt_stream.t = <abstr>
val push : '_a option -> unit = <fun>
# push (Some 1);;
- : unit = ()
# push (Some 2);;
- : unit = ()
# push (Some 3);;
- : unit = ()
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Return 1
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Return 2
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Return 3
```

```
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Sleep
```

Note that streams are consumable. Once you take an element from a stream, it is removed from it. So, if you want to iterate two times over a stream, you may consider "clonning" it, with `Lwt_stream.clone`. Cloned stream will return the same elements in the same order. Consuming one will not consume the other. For example:

```
# let s = Lwt_stream.of_list [1; 2];;
val s : int Lwt_stream.t = <abstr>
# let s' = Lwt_stream.clone s;;
val s' : int Lwt_stream.t = <abstr>
# Lwt.state (Lwt_stream.next s);;
- : int Lwt.state = Return 1
# Lwt.state (Lwt_stream.next s);;
- : int Lwt.state = Return 2
# Lwt.state (Lwt_stream.next s');;
- : int Lwt.state = Return 1
# Lwt.state (Lwt_stream.next s');;
- : int Lwt.state = Return 2
```

2.5.4 Mailbox variables

The `Lwt_mvar` module provides mailbox variables. A mailbox variable, also called a "mvar", is a cell which may contain 0 or 1 element. If it contains no elements, we say that the mvar is empty, if it contains one, we say that it is full. Adding an element to a full mvar will block until one is taken. Taking an element from an empty mvar will block until one is added.

Mailbox variables are commonly used to pass messages between threads.

Note that a mailbox variable can be seen as a pushable stream with a limited memory.

3 Running a Lwt program

Threads you create with `Lwt` always have the type `Lwt.t`. If you want to write a program and run it this is not enough. Indeed you don't know when a `Lwt` thread is terminated.

For example if your program is just:

```
let _ = Lwt_io.printl "Hello, world!"
```

you have no guarantee that the thread writing `"Hello, world!"` on the terminal will be terminated when the program exit. In order to wait for a thread to terminate, you have to call the function `Lwt_main.run`:

```
val Lwt_main.run : 'a Lwt.t -> 'a
```

This functions wait for the given thread to terminate and returns its result. In fact it does more than that; it also run the scheduler which is responsible for making thread to progress when events are received from the outside world.

So basically, when you write a `Lwt` program you must call at the toplevel the function `Lwt_main.run`. For instance:

```
let () = Lwt_main.run (Lwt_io.printl "Hello, world!")
```

Note that you must call `Lwt_main.run` only once at a time. It cannot be used anywhere to get the result of a thread. It must only be used in the entry point of your program.

4 The `lwt.unix` library

The package `lwt.unix` contains all `unix` dependent modules of `Lwt`. Among all its features, it implements cooperative versions of functions of the standard library and the unix library.

4.1 Unix primitives

The `Lwt_unix` provides cooperative system calls. For example, the `Lwt` counterpart of `Unix.read` is:

```
val read : file_descr -> string -> int -> int -> int Lwt.t
```

`Lwt_io` provides features similar to buffered channels of the standard library (of type `in_channel` or `out_channel`) but cooperatively.

`Lwt_gc` allows you to register a finaliser that returns a thread. At the end of the program, `Lwt` will wait for all the finaliser to terminate.

4.2 The Lwt scheduler

Threads doing IO may be put asleep until some events are received by the process. For example when you read from a file descriptor, you may have to wait for the file descriptor to become readable if no data are immediatly available on it.

`Lwt` contains a shceduler which is responsible for managing multiple threads waiting for events, and restart them when needed. This scheduler is implemented by the two modules `Lwt_engine` and `Lwt_main`. `Lwt_engine` is a low-level module, it provides signatures for IO multiplexers as well as several builtin implementation. `Lwt` support by default multiplexing IO with `libev` or `Unix.select`. The signature is given by the class `Lwt_engine.t`.

`libev` is used by default on Unix, because it supports any number of file descriptors while Unix.select supports only 1024 at most, and is also much more efficient. On Windows `Unix.select` is used because `libev` does not works properly. The user may change at any time the backend in use.

The engine can also be used directly in order to integrate other libraries with `Lwt`. For example `GTK` need to be notified when some events are received. If you use `Lwt` with `GTK` you need to use the `Lwt` scheduler to monitor `GTK` sources. This is what is done by the `lwt.glib` package.

The `Lwt_main` module contains the *main loop* of `Lwt`. It is run by calling the function `Lwt_main.run`:

```
val Lwt_main.run : 'a Lwt.t -> 'a
```

This function continously run the scheduler until the thread passed as argument terminates.

4.3 The logging facility

The package `lwt.unix` contains a module `Lwt_log` providing loggers. It supports logging to a file, a channel, or to the syslog daemon. You can also define your own logger by providing the appropriate functions (function `Lwt_log.make`).

Several loggers can be merged into one. Sending logs on the merged logger will send these logs to all its components.

For example to redirect all logs to `stderr` and to the syslog daemon:

```
# Lwt_log.default_logger :=
    Lwt_log.broadcast [
      Lwt_log.channel ~close_mode:`Keep ~channel:Lwt_io.stderr ();
      Lwt_log.syslog ~facility:`User ();
    ]
;;
```

Lwt also provides a syntax extension, in the package `lwt.syntax.log`. It does not modify the language but it replaces log statement of the form:

```
Lwt_log.info_f ~section "something happened: %s" msg
```

by:

```
if Lwt_log.Section.level section <= Lwt_log.Info then
  Lwt_log.info_f ~section "somethign happend: %s" msg
else
  Lwt.return ()
```

The advantages of using the syntax extension are the following:

- it checks the log level before calling the logging function, so the arguments are not computed if not needed
- debugging logs can be removed at parsing time

By default, the syntax extension removes all logs with the level `debug`. To keep them, pass the command line option `-lwt-debug` to camlp4.

5 The Lwt.react library

The `Lwt_react` module provides helpers for using the `react` library with `Lwt`. It extends the `React` module by adding `Lwt` specific functions. It can be used as a replacement of `React`. For example you can add at the beginning of you program:

```
open Lwt_react
```

instead of:

```
open React
```

or:

```
module React = Lwt_react
```

Among the added functionalities we have `Lwt_react.E.next`, which takes an event and returns a thread which will wait until the next occurence of this event. For example:

```
# open Lwt_react;;
# let event, push = E.create ();;
val event : '_a React.event = <abstr>
val push : '_a -> unit = <fun>
# let t = E.next event;;
val t : '_a Lwt.t = <abstr>
# Lwt.state t;;
- : '_a Lwt.state = Sleep
# push 42;;
- : unit = ()
# Lwt.state t;;
- : int Lwt.state = Return 42
```

Another interesting feature is the ability to limit events (resp. signals) from occuring (resp. changing) too often. For example, suppose you are doing a program which displays something on the screeen each time a signal changes. If at some point the signal changes 1000 times per second, you probably want not to render it 1000 times per second. For that you use `Lwt_react.S.limit`:

```
val limit : (unit -> unit Lwt.t) -> 'a React.signal -> 'a React.signal
```

`Lwt_react.S.limit f signal` returns a signal which varies as `signal` except that two consecutive updates are separeted by a call to `f`. For example if `f` returns a thread which sleep for 0.1 seconds, then there will be no more than 10 changes per second. For example:

```
open Lwt_react

let draw x =
  (* Draw the screen *)
  ...

let () =
  (* The signal we are interested in: *)
  let signal = ... in

  (* The limited signal: *)
  let signal' = S.limit (fun () -> Lwt_unix.sleep 0.1) signal in

  (* Redraw the screen each time the limited signal change: *)
  S.notify_p draw signal'
```

6 The lwt.text library (deprecated)

WARNING: the `lwt.text` library is deprecated. It has been replaced by the `lambda-term` library which is more complete and more portable. It is available here: http://lambda-term.forge.ocamlcore.org/.

The `lwt.text` library provides functions to deal with text mode (in a terminal). It is composed of the three following modules:

- `Lwt_text`, which is the equivalent of `Lwt_io` but for unicode text channels
- `Lwt_term`, providing various terminal utilities, such as reading a key from the terminal
- `Lwt_read_line`, which provides functions to input text from the user with line editing support

6.1 Text channels

A text channel is basically a byte channel with an encoding. Input (resp. output) text channels decode (resp. encode) unicode characters on the fly. By default, output text channels use transliteration, so they will not fail because text you want to print cannot be encoded in the system encoding.

For example, with you locale sets to "C", and the variable `name` set to "Jérémie", you got:

```
# lwt () = Lwt_text.printlf "My name is %s" name;;
My name is J?r?mie
```

6.2 Terminal utilities

The `Lwt_term` allow you to put the terminal in *raw mode*, meaning that input is not buffered and character are returned as the user types them. For example, you can read a key with:

```
# lwt key = Lwt_term.read_key ();;
val key : Lwt_term.key = Lwt_term.Key_control 'j'
```

The second main feature of `Lwt_term` is the ability to print text with styles. For example, to print text in bold and blue:

```
# open Lwt_term;;
# lwt () = printlc [fg blue; bold; text "foo"];;
foo
```

If the output is not a terminal, then `printlc` will drop styles, and act as `Lwt_text.printl`.

6.3 Read-line

`Lwt_read_line` provides a full featured and fully customisable read-line implementation. You can either use the high-level and easy to use `read_*` functions, or use the advanced `Lwt_read_line.Control.read_*` functions.

For example:

```
# open Lwt_term;;
# lwt l = Lwt_read_line.read_line ~prompt:[text "foo> "] ();;
foo> Hello, world!
val l : Text.t = "Hello, world!"
```

The second class of functions is a bit more complicated to use, but allow to control a running read-line instance. For example you can temporary hide it to draw something, you can send it commands, fake input, and the prompt is a signal so it can change dynamically.

7 Other libraries

7.1 Detaching computation to preemptive threads

It may happen that you want to run a function which will take time to compute or that you want to use a blocking function that cannot be used in a non-blocking way. For these situations, `Lwt` allow you to *detach* the computation to a preemptive thread.

This is done by the module `Lwt_preemptive` of the `lwt.preemptive` package which maintains a pool of system threads. The main function is:

```
val detach : ('a -> 'b) -> 'a -> 'b Lwt.t
```

`detach f x` will execute `f x` in another thread and asynchronously wait for the result.

If you have to run `Lwt` code in another thread, you must use the function `Lwt_preemptive.run_in_main`:

```
val run_in_main : (unit -> 'a Lwt.t) -> 'a
```

It works as follow:

- it sends the function to the main thread and wait
- the main thread execute the function

- when it terminates the main thread sends back the result
- the result is returned

Note that you cannot call `Lwt_main.run` in another system thread, so you must use this function.

7.2 SSL support

The package `lwt.ssl` provides the module `Lwt_ssl` which allow to use SSL asynchronously

7.3 Glib integration

The `lwt.glib` embeds the `glib` main loop into the `Lwt` one. This allows you to write GTK application using `Lwt`. The one thing you have to do is to call `Lwt_glib.install` at the beginning of you program.

8 Writing stubs using `Lwt`

8.1 Thread-safe notifications

If you want to notify the main thread from another thread, you can use the `Lwt` thread safe notification system. First you need to create a notification identifier (which is just an integer) from the OCaml side using the `Lwt_unix.make_notification` function, then you can send it from either the OCaml code with `Lwt_unix.send_notification` function, or from the C code using the function `lwt_unix_send_notification` (defined in `lwt_unix_.h`).

Notifications are received and processed asynchronously by the main thread.

8.2 Jobs

For operations that can not be executed asynchronously, `Lwt` uses a system of jobs that can be executed in a different threads. A job is composed of three functions:

- A stub function to create the job. It musts allocate a new job structure and fill its [worker] and [result] fields. This function is executed in the main thread. The return type for the OCaml external must be of the form `'a job`.
- A function which executes the job. This one may be executed asynchronously in another thread. This function must not:
  - access or allocate OCaml block values (tuples, strings, ...),
  - call OCaml code.
- A function which reads the result of the job, free resources and return the result as an OCaml value. This function is executed in the main thread.

With `Lwt < 2.3.3`, 4 functions (including 3 stubs) were required. It is still possible to use this mode but it is deprecated.

We show as example the implementation of `Lwt_unix.mkdir`. On the C side we have:

```
/* Structure holding informations for calling [mkdir]. */
struct job_mkdir {
  /* Informations used by lwt.
     It must be the first field of the structure. */
  struct lwt_unix_job job;
  /* This field store the result of the call. */
```

```
  int result;
  /* This field store the value of [errno] after the call. */
  int errno_copy;
  /* Pointer to a copy of the path parameter. */
  char* path;
  /* Copy of the mode parameter. */
  int mode;
  /* Buffer for storing the path. */
  char data[];
};

/* The function calling [mkdir]. */
static void worker_mkdir(struct job_mkdir* job)
{
  /* Perform the blocking call. */
  job->result = mkdir(job->path, job->mode);
  /* Save the value of errno. */
  job->errno_copy = errno;
}

/* The function building the caml result. */
static value result_mkdir(struct job_mkdir* job)
{
  /* Check for errors. */
  if (job->result < 0) {
    /* Save the value of errno so we can use it
       once the job has been freed. */
    int error = job->errno_copy;
    /* Copy the contents of job->path into a caml string. */
    value string_argument = caml_copy_string(job->path);
    /* Free the job structure. */
    lwt_unix_free_job(&job->job);
    /* Raise the error. */
    unix_error(error, "mkdir", string_argument);
  }
  /* Free the job structure. */
  lwt_unix_free_job(&job->job);
  /* Return the result. */
  return Val_unit;
}

/* The stub creating the job structure. */
CAMLprim value lwt_unix_mkdir_job(value path, value mode)
{
  /* Get the length of the path parameter. */
  mlsize_t len_path = caml_string_length(path) + 1;
  /* Allocate a new job. */
  struct job_mkdir* job =
    (struct job_mkdir*)lwt_unix_new_plus(struct job_mkdir, len_path);
  /* Set the offset of the path parameter inside the job structure. */
  job->path = job->data;
```

```c
  /* Copy the path parameter inside the job structure. */
  memcpy(job->path, String_val(path), len_path);
  /* Initialize function fields. */
  job->job.worker = (lwt_unix_job_worker)worker_mkdir;
  job->job.result = (lwt_unix_job_result)result_mkdir;
  /* Copy the mode parameter. */
  job->mode = Int_val(mode);
  /* Wrap the structure into a caml value. */
  return lwt_unix_alloc_job(&job->job);
}
```

and on the ocaml side:

```ocaml
(* The stub for creating the job. *)
external mkdir_job : string -> int -> unit job = "lwt_unix_mkdir_job"

(* The ocaml function. *)
let mkdir name perms = Lwt_unix.run_job (mkdir_job name perms)
```