Mirage Syntax Extensions

September 16, 2011 — OCamlPro SAS (**http://www.ocamlpro.com/**)

Lwt

Mirage contains a large subset of the *Core* Lwt library. As the underlying OS interface is not POSIX compatible, the Lwt Unix bindings, the Lwt Terminal manipulation and Lwt Miscellaneous libraries are not included in Mirage.

Base functions

```
type 'a t     (* A thread returning a value of type α *)
val return    : 'a -> 'a t
val bind      : 'a t -> ('a -> 'b t) -> 'b t
val join      : unit t list -> unit t
val choose    : 'a t list -> 'a t
val map       : ('a -> 'b) -> 'a t -> 'b t
```

Operators

Operators	Equivalent functions
`t >= fun x -> f x`	`bind t (fun x -> f x)`
`lwt x = t in f x`	`bind t (fun x -> f x)`
`lwt x = t1 and y = t2 in f`	`bind t1 (fun x ->` `  bind t2 (fun y -> f))`
`t1 >> t2`	`bind t1 (fun _ -> t2)`
`t1 <&> t2 <&> t3`	`join [t1;t2;t3]`
`t1 <?> t2 <?> t3`	`choose [t1;t2;t3]`
`t >|= f`	`map f t`

Exceptions

```
try_lwt
  (expr:'a t)
```

```
try_lwt
  (expr:'a t)
finally
  (expr:'b t)
```

```
try_lwt
  (expr:'a t)
with
  | (patt1:exn) -> (expr1:'a t)
  | (patt2:exn) -> (expr2:'a t)
```

```
try_lwt
  (expr:'a t)
with
  | (patt1:exn) -> (expr1:'a t)
  | (patt2:exn) -> (expr2:'a t)
finally
  (expr:'b t)
```

Iterations

```
for_lwt i = 0 to 10 do
  (expr:unit t)
done
```

```
while_lwt (expr:bool) do
  (expr:unit t)
done
```

```
for_lwt i in (expr:'a Lwt_stream.t) do
  (expr:unit t)
done
```

Lwt pattern-matching

```
match_lwt (expr:'a t) with
  | (patt1:'a) -> (expr1:'b)
  | (patt2:'a) -> (expr2:'b)
```

COW

Mirage contains syntax extensions to write web documents within OCaml programs.

XML & XHTML

The only difference between XML and XHTML in Mirage is that special HTML entities are escaped in XHTML. The both APIs are almost identical otherwise.

Quotations

```
let xhtml : Html.t =
  <:html< <h1>This is a title</h1> >>
let xml : Xml.t =
  <:xml< <tag>this is a tag</tag> >>
```

Escaping

```
<:html< <script><!CDATA[
  ....
]]></script> >>
```

Templates

```
let str = "This is template with a $title$"
let xhtml =
 Html.of_string ~template:["title", xhtml] str
```

Pretty-printing

```
let str = Html.to_string (expr:Html.t)
let str = Xml.to_string (expr:Xml.t)
```

Antiquotations (ie. typed templates)

```
let x = (expr:Html.t) in <:html< ... $x$ ... >>
let x = (expr:Html.t option) in <:html< $opt:x$ >>
let x = (expr:int) in <:html< $int:x$ >>
let x = (expr:float) in <:html< $float:x$ >>
let x = (expr:string) in <:html< $str:x$ >>
let x = (expr:Html.t list) in <:html< $list:x$ >>

let x = [ ("class", "bar") ] in
 <:html< <div foo $alist:x$ /> >>
let x = "class=bar id=foo" in
 <:html< <div foo $attrs:x$ /> >>
```

These are both expanded to `<div foo class="bar" />`

HTML API

```
type link         = { text: string; href: string}
val html_of_link  : link -> Html.t
val nil           : Html.t
val interleave    : string array -> Html.t list -> Html.t
type table        = Html.t array array
val html_of_table : ?headings:bool -> table -> Html.t
```

CSS

CSS quotations are almost valid CSS fragments, with the following exceptions :

- Property name and values MUST be separated by `' '`
- each proporty line MUST ends with `';'`
- selectors CAN be separated by `';'`
- selector CAN be nested

Quotations

```
let expr : Css.t =
 <:css< 1px solid black >>
let prop : Css.t =
 <:css< border: 1px solid black; >>
let selector : Css.t =
 <:css< h1 { border: 1px solid black; } >>
```

Pretty-printing

```
let str = Css.to_string (expr:Css.t)
```

Antiquotations

```
let color1 = <:css< #fff >>
and color2 = <:css< #e3e3e3 >> in
<:css< color: blue $expr:[color1; color2]$ >>

let css = (expr:Css.t) in
<:css< body { $css$; color: blue; } >>

let css_l = (expr:Css.t list) in
<:css< body { $prop:css_l$; color: blue; } >>
```

The COW syntax extension can automatically unfold nested selectors. The two expressions are equivalent :

Nested	Unfolded

`let css = <:css<`
`  body {`
`   color: blue;`
`   h1 {`
`    color: red;`
`   }`
`  }`
`>>`
	`let css = <:css<`
	`  body {`
	`   color: blue;`
	`   }`
	`  body h1 {`
	`   color: red;`
	`  }`
	`>>`

CSS API

```
val rounded        : Css.t   Rounded box
val top_rounded    : Css.t   Top-rounded box
val bottom_rounded : Css.t   Bottom-rounded box
val box_shadow     : Css.t   Add box shadow
val text_shadow    : Css.t   Add text shadow
val reset_padding  : Css.t   Reset margins and padding
```