The Ml-Like for the web Documentation & Manual

Contents

Introduction

Ml-Like for the web is a functional language for writing dynamic webpages, evaluated on the server side.

Language Syntax

Webpage

A *dynamic webpage* (i.e. the programs we evaluate) consists of a list of *elements*.

Each element can be of three sorts:
- pure HTML code ; or
- Ml-Like for the web code, which can be either:
  - a list of *global declarations* ; or
  - an expression.

```
bodyexp: anyHtmlCode (<{exp | (global_declaration)*}> anyHtmlCode)*
```

Global declaration

A global declaration declares and defines a variable, whose scope is (the remainder of) the entire webpage. They (currently) are of two kinds: either a simple global declaration (with `let`) or a declaration of a session variable `Session.let`.

```
declarator: let | Session.let
```

```
global_declaration: declarator id = exp
```

For now, only expressions can be declared by the user. At some point, we would like to allow user-defined types and modules.

Expressions

General expressions:

```
exp: e, e', e'', … ::=
  | let id = e in e'
  | fun id -> e
  | fixfun id id -> e
  | e e'
  | if e then e' else e''
  | e;e'
  | id
  | aexp | bexp | sexp
  | texp
  | uexp
  | html | dbexp
  | (e)
  | begin e end
```

Arithmetic expressions:

```
aexp:
  | exp + exp
  | exp - exp
  | exp * exp
  | exp / exp
  | exp ^ exp
  | int_literal
```

Boolean expressions:

```
bexp:
  | exp < exp
  | exp > exp
  | exp <= exp
  | exp >= exp
  | exp = exp
  | exp <> exp
  | exp && exp
  | exp || exp
  | not exp
  | bool_literal
```

String expressions:

```
sexp: exp ++ exp | string_literal
```

Tuple expressions:

```
texp: exp, exp
```

For now, only couples are allowed, and `(x1, x2, x3, x4)` is parsed as `(x1, (x2, (x3, x4)))`.

HTML:

```
html: <[ anyHtmlCode ]>
```

Identifiers

In Ml-Like for the web, identifers are *namespaced*. An identifier is a sequence of namespace and the actual variable name.

```
value_name: (_|[a-z])(_|'|[0-9]|[a-z]|[A-Z])*
```

Whereas a path is a sequence of modules name separated by a dot:

`module_name : [A-Z](_|'|[0-9]|[a-z]|[A-Z])*`

`path: M, M' ::= ***module_name*** | ***module_name***.M'`

A identifier (path to a value) is then:

`id: (***path***.)?***value_name***`

Example.
- variable
- Session.my_function_n_362
- _MyFunction
- A.B.C.D.E.myVariable067

But not:
- MyFunction
- SomeModule.01var

Literals

Integers

For better readability for the programmer, we allow underscores in numbers.

`int_literal: [0-9]([0-9]|_)*`

Example.
- 123
- 100_000
- 1_2_____3____

Strings

`string_literal: "…"`

Booleans

`bool_literal: true, false`

Type system

Ml-Like for the web is strongly, statistically typed. For now, user type annotation are not allowed.

Types

The only atomic native types represent integers, booleans, strings, unit, html code and databases. They can be combined with type constructors $\longrightarrow$ for functions and $\times$ for tuples. For now, sum types are not implemented, but should be with user-defined types.

`tlit: int | bool | string | unit | html | db`

`α, β, … ::= α -> β | α times β`

Typing rules

The only lax rule is the typing rule of sequences: if the left handside of the `;` is not of type unit, it will only raise a warning.

$$\frac{\Gamma \vdash \texttt{e} : \alpha \quad \Gamma, \texttt{x} : \alpha \vdash \texttt{e`} : \beta}{\Gamma \vdash \texttt{let x = e in e`} : \beta}$$

$$\frac{\Gamma \vdash \texttt{e} : \alpha \quad \Gamma, \texttt{x} : \alpha \vdash \texttt{e`} : \beta}{\Gamma \vdash \texttt{let x = e in e`} : \beta} \qquad \frac{\Gamma, \texttt{x} : \alpha \vdash \texttt{e} : \beta}{\Gamma \vdash \texttt{fun x -> e} : \alpha \to \beta} \qquad \frac{\Gamma, \texttt{f} : \alpha \to \beta, \texttt{x} : \alpha \vdash \texttt{e} : \beta}{\Gamma \vdash \texttt{fixfun f x -> e} : \alpha \to \beta}$$

$$\frac{\Gamma \vdash \texttt{e} : \alpha \to \beta \quad \Gamma \vdash \texttt{e`} : \alpha}{\Gamma \vdash \texttt{e e`} : \beta} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{bool} \quad \Gamma \vdash \texttt{e`} : \alpha \quad \Gamma \vdash \texttt{e``} : \alpha}{\Gamma \vdash \texttt{if e then e` else e``} : \alpha} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{unit} \quad \Gamma \vdash \texttt{e`} : \alpha}{\Gamma \vdash \texttt{e;e`} : \alpha}$$

$$\circledast\texttt{: +, -, *, /, or ^} \ \frac{\Gamma \vdash \texttt{e} : \texttt{int} \quad \Gamma \vdash \texttt{e`} : \texttt{int}}{\Gamma \vdash \texttt{e} \circledast \texttt{e`} : \texttt{int}} \qquad \circledast\texttt{: >, <, >=, <=, = or <>} \ \frac{\Gamma \vdash \texttt{e} : \alpha \quad \Gamma \vdash \texttt{e`} : \alpha}{\Gamma \vdash \texttt{e} \circledast \texttt{e`} : \texttt{bool}}$$

$$\circledast\text{: \&\& or ||} \ \frac{\Gamma \vdash \texttt{e} : \texttt{bool} \quad \Gamma \vdash \texttt{e`} : \texttt{bool}}{\Gamma \vdash \texttt{e} \circledast \texttt{e`} : \texttt{bool}} \qquad \frac{\Gamma \vdash \texttt{e} : \texttt{bool}}{\Gamma \vdash \texttt{not e} : \texttt{bool}} \qquad \frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha}$$

$$\frac{}{\Gamma \vdash \texttt{b} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{n} : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{<(f)string literal>} : \texttt{string}} \qquad \frac{}{\Gamma \vdash \texttt{<[html code]>} : \texttt{html}}$$

$$\frac{\Gamma \vdash \texttt{e} : \alpha \quad \Gamma \vdash \texttt{e`} : \beta}{\Gamma \vdash \texttt{(e, e`)} : \alpha \times \beta} \qquad \frac{}{\Gamma \vdash \texttt{fst} : \alpha \times \beta \to \alpha} \qquad \frac{}{\Gamma \vdash \texttt{snd e} : \alpha \times \beta \to \beta} \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash \texttt{sqlite\_opendb} : \texttt{string -> db}} \qquad \frac{}{\Gamma \vdash \texttt{sqlite\_closedb} : \texttt{db -> bool}}$$

$$\frac{}{\Gamma \vdash \texttt{sqlite\_exec} : \texttt{db -> (html -> html -> html) -> (html -> string -> string -> html) -> bool}}$$

Program semantics

Values

`values: v, v', … ::= ⟨E, ***function***⟩ | n | true | false | ***string_literal*** | (v, v') | | pure_html_code | evald_page`

`function: fun x -> e | fixfun x -> e`

`evald_page: [v1; v2; …; vn]` ⚠️ it's *not* Ml-Like for the web list.

`vdb: a value representing a database in the language`

An evaluated webpage can be injected in a value (via the frame).

Example. This happens when we evaluate `<{ <[htmlcode <{let x = 1 in x}> somemorehtmlcode]> }>`

Evaluation rules

A dynamic webpage to evaluate is seen as a list of either:

- Pure html code ;
- An expression ;
- A global declaration.

The interpreter evaluates following a big-step call-by-value semantics.

Expression

Functions

Functions are values; when they evaluate the variables needed to their evaluation are automatically captured. Functions can be passed as argument to other functions.

To write recursive functions, you must use `fixfun` it binds two variables: the first one is the name of the function (for the recursive call), the second one is the name of the argument.

Example. The following code represents the factorial function:

```
fixfun fact n -> if n = 0 then 1 else n * fact (n - 1)
```

⚠️ if you declare a function, you mustn't mistake the name of the resulting function in the let-binding and the name used for recursing. For instance:

```
let fact = fixfun fact n -> if n = 0 then 1 else n * fact (n-1)
let fact = fixfun f n -> if n = 0 then 1 else n * f (n-1)
```

both defines `fact` as the factorial function. Although:

```
let fact = fixfun f n -> if n = 0 then 1 else n * fact (n-1)
```

is incorrect, and will raised an error because `fact` is not defined when used here.

There is currently no support for mutually recursive functions.

Declarations

A declaration can either be *global* i.e. its scope is the remainder of the file or *local* i.e. its scope is restricted to the expression after the `in` keyword.

Furthermore, global session declarations have a specific semantics: `Session.let x = e` will firstly have the same effect as `let x = e`, except:

- the bound variable is stored within the `Session` module i.e. to access it somewhere else, we need to write `Session.x`;
- it defines a *session variables* i.e. it will be accessible across all the pages from now on, for the duration of the session (see bullet point Session Cookie).

Operators

The language provides some operators on basic datatypes:

- And/or/not on booleans: `&&`, `||`, `not`.
- Addition, multiplication, substraction, division, exponentiaion: `+`, `*`, `-` ,`/` ,`^`. The unary version of `-` is also available.
- comparison operators less than, greater than, less or equal than, greater or equal than, equal, not equal: `<`, `>`, `<=`, `>=`, `=`, `<>`. These are *polymorphic* i.e. they typecheck for any two expressions, provided that they are of the same type. It will fail at execution and raise an error on types for which it's not implemented. Comparison is currently implemented only for basic types: integers, booleans, strings and *content* i.e. Ml-Like for the web code evaluated to HTML.
- Concatenation of two strings: `++`.

Predefined functions

Projections

To get the first or the second element of a *pair*, `fst` and `snd` are pre-defined.

Arguments from the command line

You can have pre-defined values before even starting to evaluate the page.

To do that, you can pass them directly in the command line:

```
Usage: <program> <source> <dest> [<argoption> <arguments>]*
  <source>: can either be a path to a file, or -stdin.
  <dest>: can either be a path to a file, or -stdout.
  You can pass argument to the program to pre-load a variable environment before evaluating the dynamic
webpage.
  <argoption>: the way the program should interpret the arguments immediatly following this flag. Can be
any of the following options:
    -argrepr: interpret each value following the built-in repr function. This allows to pass argument of
any type.
```

```
    -argstr: interpret each value as a string.
  <arguments>: A list of bindings, of th following form: METHOD&name1=arg1&...&namen=argn.
    These variable will be available in the dynamic webpage within the namespace Method.
```

This will create a module containing all the defined variables.

Example. `<program> file -stdout -argstr MODULE&x=astring&y=anotherstring` will evaluate `file`, but if `file` uses variable `Module.x` (resp. `Module.y`), it will be globally defined and its value will be `"astring"` (resp. `"anotherstring"`).

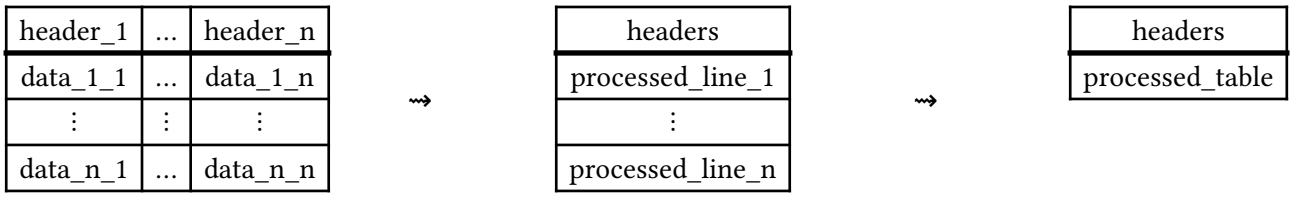
Database operations

Three built-in functions are provided to interact with databases using SQLite via the sqlite3 ocaml library.

One function opens a database, one to close one and one to perform SQL on them. All these functions are available in the module `Sqlite`.

`Sqlite.open "path/to/adb"` evaluates to a value representing the database at path `path/to/db`, with which we can now interact using `exec`.

`Sqlite.exec vdb fl fc query = processed_table` where `vdb` is a value obtained from `open`. `query` is a string corresponding to a SQL query, which is executed on the database `vdb`. If it has query statements, then `sqlite_exec` allow to process the resulting table with a double *left-folding* function applied on the resulting table i.e. let the resulting table be:



Firstly, `fc` is used to *fold* each line into a single HTML value `processed_line_i`. More precisely,

`processed_line_i = fc (... (fc EmptyHtmlCode header_1 data_i_1) ...) header_n data_i_n`.

And now, we then again fold the resulting column into a single HTML value, using the `fl` function:

`processed table = fl (... (fl EmptyHtmlCode line_1) ...) line_n`.

Example. If you want to get the resulting relation of SQL query `query` on `vdb` as a HTML table, you can write the following code:

```
<table>
<{
  Sqlite.exec vdb
    (fun acc -> fun new_line -> <[ <{acc}> <tr> <{new_line}> </tr> ]>)
    begin fun acc -> fun hd -> fun content ->
      <[
        <{ acc }>
        <td> <{content}> </td>
      ]>
    end query
}>
</table>
```

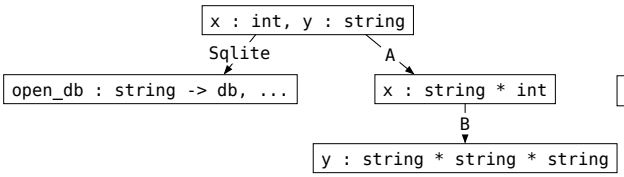
`Sqlite.close vdb` closes the databases `vdb`. Returns a boolean telling whether the database could be closed. If `false`, the the database *remains open*. See here for reason the database couldn't be closed, and more specifications of the sqlite functions.

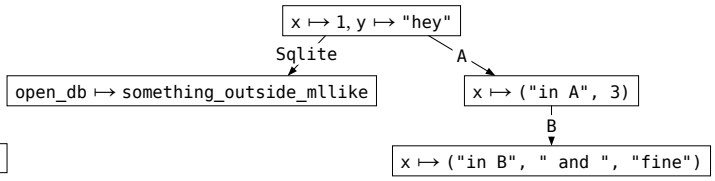
Namespacing, modules and session variables

At some point, we would like to implement modules contained inside another module, maybe even functors if possible, and records types. When this is done, a full variable name will be `((modulevar.)*(expr.)*varname)` where each `expr` must be of type record.

For instance, if module A contains module B which itself contains a record `r : {r' : rec'}` where `rec'` itself is a record with a field `x`, then `A.B.r.r'.x` designates the field `x` of the farthest nested record.

A modular typing (resp. evaluation) environment therefore becomes a tree, where each edge is labelled with a module name, and each node is labelled with a typing (resp. evaluation) environment.


x : int, y : string
Sqlite
A
open_db : string -> db, ...
x : string * int
B
y : string * string * string


x ↦ 1, y ↦ "hey"
Sqlite
A
open_db ↦ something_outside_mllike
x ↦ ("in A", 3)
B
x ↦ ("in B", " and ", "fine")

TODO

☐ Implement escape characters in strings

☐ Implement fstrings

☑️ Implement percent-encoding

☐ Don’t lex ml located in html comment

☐ Add comments within ML

☑️ Allow HTML brackets to contain any dynpage e.g. <[somehtml <{“coucou”}> somemorehtml]>

☐ Add syntactic sugar for multiple variables functions.

☐ Add t-uples

☐ Add pattern-matching

☑️ Add superglobal variables

☑️ Add user-defined global variables

☐ At some point, we would like to implement modules contained inside another module, and records types.

☐ Add user-defined types

☐ Once it’s done, implement basic types such as list directly within the language.

☐ Allow type annotations from the user

☐ Allow importing other ml files (as modules ?)

☐ Keep line number information on parsed term for better typing error messages (?)