Tierless Web programming in ML

Gabriel RADANNE
An HTTP Request

GET /hypertext/WWW/TheProject.html
HTTP/1.1
Host: info.cern.ch
User-Agent: Firefox/56.0
Accept: text/html
Accept-Language: en
Accept-Encoding: gzip, deflate
Referer: http://info.cern.ch/
World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists, Policy, November’s W3 news, Frequently Asked Questions.

What's out there?
Pointers to the world's online information, subjects, W3 servers, etc.

Help
on the browser you are using

Software Products
A list of W3 project components and their current state. (e.g. Line Mode, X11 Viola, NeXTStep, Servers, Tools, Mail robot, Library)

Technical
Details of protocols, formats, program internals etc

Bibliography
Paper documentation on W3 and references.

People
A list of some people involved in the project.

History
A summary of the history of the project.

How can I help?
If you would like to support the web.

Getting code
Getting the code by anonymous FTP, etc.
Client

Server

Client

Server
Join the numbers and get to the 2048 tile!
Welcome to my defense!
Welcome to my defense!
Welcome to my defense!
Welcome to my defense!
Server Send

line 1: Welcome to my defense!

Client Expect

line <number>: <text>
Server Send
1,0: Welcome to my defense!

Client Expect
line <number>: <text>
One program for everything

Client ↔ Server

① ②
One program for everything

Client

Server

Tierless languages
The OCSIGEN project
The OCSIGEN project

**ELIOM**

**SERVER**

**JS_OF_OCAML**

**OCAML**
The OCSIGEN project

Libraries

Language extension

SERVER | JS_OF_OCAML

OCAML
The OCSIGEN project

Libraries

Language extension

SERVER  JS_OF_OCAML

OCAML
Location annotations allow to use client and server code *in the same program.*

```plaintext
1  type %client t = ...
2  let %server v = ...
```

The program is statically sliced during compilation.
Building fragments of client code inside server code

Fragments of client code can be included inside server code.

```mermaid
let%server x : int fragment = [%client 1 + 3 ]
```
Building fragments of client code inside server code

Fragments of client code can be included inside server code.

```ocaml
let%server x : int fragment = [%client 1 + 3 ]
let%server y = [ ("foo", x) ; ("bar", [%client 2]) ]
```
Accessing server values in the client

Injections allow to use server values on the client.

```plaintext
let\%server s : int = 1 + 2
let\%client c : int = ~\%s + 1
```
Everything at once

We can combine injections and fragments.

```
1 let server x : int fragment = [%client 1 + 3 ]
2 let client c : int = 3 + ~%x
```
Small example – Hint button

```elixir
let%server hint_button (msg : string) =
    button
    ~a:[a_onclick [%client fun _ -> alert ~%msg ] ]
    [pcdata "Show Hint"]
```

```html
<button onclick="...">Show hint</button>
```
let%server hint_button (msg : string) =
  button
    ~a:[a_onclick[%client fun _ -> alert ~%msg ]]
    [pcdata "Show Hint"]

<button onclick="...">
  Show hint
</button>
Before my thesis

The ELIOM “language” was already implemented as an OCAML syntax extension by numerous contributors:

- Vincent BALAT
- Benedikt BECKER
- Pierre CHAMBART
- Grégoire HENRY
- Vasilis PAPAVASILIEOU
- Jérôme VOUILLON

Problem

The language was starting to get big and there was no formal definition.
Before my thesis

The ELIO M “language” was already implemented as an OCAML syntax extension by numerous contributors:

- Vincent BALAT
- Benedikt BECKER
- Pierre CHAMBART
- Grégoire HENRY
- Vasilis PAPAVASILIEOU
- Jérôme VOUILLON

Problem

The language was starting to get big and there was no formal definition.
My contributions

• A formalization of the type system, the semantics and the compilation scheme
• Improvements on the ELIOM language
  • New type system defined as an extension of the OCAML one
  • New module system
• A new implementation which closely reflects the formalization
1. Formalization
   - Semantics
   - Compilation

2. Type system

3. Module system
Small example

```ocaml
let%server hint_button (msg : string) =
  button
    ~a:[a_onclick [%client fun _ -> alert ~%msg ] ]
    [pcdata "Show hint"]

let%server thebutton = hint_button "Boo!"
```

How is that actually executed?
Small example

```ocaml
let%server hint_button (msg : string) =
  button
  ~a:[a_onclick [%client fun _ -> alert ~%msg ] ]
  [pcdata "Show hint"]

let%server thebutton = hint_button "Boo!"
```

How is that actually executed?
Example of execution

**ELIOM program**

```eliom
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

**Client program**

**ELIOM environment**

**Client environment**
Example of execution

**ELIOM program**

```eliom
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

**Client program**

```eliom
let f () = 1 + 3
```
Example of execution

**ELIOM program**

```plaintext
let%server x = r
let%client y = 3 + ~%x
return y
```

**Client program**

```plaintext
let f () = 1 + 3
let r = f ()
```
Example of execution

ELIOM program

```ml
let%client y = 3 + ~%x
return y
```

ELIOM environment

```
x ↦ r
```

Client program

```ml
let f () = 1 + 3
let r = f ()
```

Client environment

-
Example of execution

**ELIOM program**

```elm
let\%client y = 3 + r
return y
```

**Client program**

```elm
let f () = 1 + 3
let r = f ()
```

**ELIOM environment**

```elm
x \mapsto r
```
Example of execution

ELIOM program

```
return y
```

ELIOM environment

```
x \mapsto r
```

Client program

```
let f () = 1 + 3
let r = f ()
let y = 3 + r
```
Example of execution

```
let f () = 1 + 3
let r = f ()
let y = 3 + r
return y
```
Example of execution

**ELIOM program**

```
let f () = 1 + 3
let r = f ()
let y = 3 + r
return y
```

**ELIOM environment**

```
x → r
```

**Client program**

```
let f () = 1 + 3
let r = f ()
let y = 3 + r
return y
```

**Client environment**
Example of execution

ELIOM program

Example of execution

ELIOM environment

x → r

Client program

let r = f ()
let y = 3 + r
return y

Client environment

f → fun() -> 1+3
Example of execution

ELIOM program

Client program

let y = 3 + r
return y

ELIOM environment

Client environment

x $\mapsto$ r

Client program

let y = 3 + r
return y

Client environment

f $\mapsto$ fun() -> 1+3
r $\mapsto$ 4
Example of execution

ELIOM program

\[ x \rightarrow r \]

Client program

\texttt{return y}

Client environment

\[
\begin{align*}
\texttt{f} & \rightarrow \texttt{fun()} - \rightarrow 1 + 3 \\
\texttt{r} & \rightarrow 4 \\
y & \rightarrow 7
\end{align*}
\]
Example of execution

**ELIOM program**

- $x \mapsto r$

**ELIOM environment**

- $x \mapsto r$

**Client program**

- $f \mapsto \text{fun()} \rightarrow 1+3$

**Client environment**

- $f \mapsto \text{fun()} \rightarrow 1+3$
- $r \mapsto 4$
- $y \mapsto 7$

**Result**

- $7$
Example of compilation

Eliom code

```
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

Client code

```
bind "f0" (fun () -> 1 + 3);
exec ();
let y = 3 + get "i"
return y
```

Server code

```
let x = fragment "f0" ()
end ();
inject "i" x;
```
Execution of the compiled code

Server program

```ocaml
let x = fragment "f0" ();
end ();
inject "i" x;
```

ELIOM program

```ocaml
let%server x =
  [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

Queue

Client program

```ocaml
bind "f0" (fun()->1+3);
exec ();
let y = 3 + get "i"
return y
```

Server environment

Client environment

Injections
Execution of the compiled code

Server program

```plaintext
let x = "r"
end ();
inject "i" x;
```

Queue

```plaintext
"r" \mapsto "f0]()
```

Client program

```plaintext
bind "f0" (fun()\rightarrow 1+3);
exec();
let y = 3 + get "i"
return y
```

ELIOM program

```plaintext
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

Server environment

Client environment

Injections
Execution of the compiled code

**Server program**

```
end ();
inject "i" x;
```

**Queue**

```
"r" \mapsto "f0"()
```

**Server environment**

- `x \mapsto "r"`

**ELIOM program**

```
let %server x = 
  [%client 1 + 3]
let %client y = 3 + ~%x
return y
```

**Client environment**

```
bind "f0" (fun()->1+3);
exec ();
let y = 3 + get "i"
return y
```

**Client program**
Execution of the compiled code

**Server program**

\[
\text{inject "i" x;}
\]

**Server environment**

\[x \mapsto "r"\]

**Queue**

\["r" \mapsto "f0"()\]

**Client program**

\[
\text{bind "f0" (fun()->1+3); exec (); let y = 3 + get "i"; return y}
\]

**ELIOM program**

\[
\text{let%server } x = [\text{client 1 + 3}]
\text{let%client } y = 3 + ~%x
\text{return y}
\]
Execution of the compiled code

**ELIOM program**

```eliom
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

**Server program**

```
x → "r"
```

**Queue**

```
"r" → "f0"()
```

**Injections**

```
"i" → "r"
```

**Client program**

```
bind "f0" (fun() -> 1+3);
exec();
let y = 3 + get "i"
return y
```
Execution of the compiled code

Server program

x → "r"

Server environment

ELIOM program

let%server x =
  [%client 1 + 3]
let%client y = 3 + ~%x
return y

Queue

"r" ↦ "f0"()
end

Injections

"i" ↦ "r"

Client environment

Client program

bind "f0" (fun()->1+3);
exec();
let y = 3 + get "i"
return y
Execution of the compiled code

Server program

Server environment

x \mapsto "r"

Queue

"r" \mapsto "f0"()

Injections

"i" \mapsto "r"

Client environment

"f0" \mapsto \text{fun()}-\rightarrow 1+3

Client program

let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y

exec ();
let y = 3 + get "i"
return y
Execution of the compiled code

ELIOM program

```
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

Server program

<table>
<thead>
<tr>
<th>x</th>
<th>&quot;r&quot;</th>
</tr>
</thead>
</table>

Server environment

Queue

```
exec ();
let y = 3 + get "i"
return y
```

Client program

<table>
<thead>
<tr>
<th>&quot;f0&quot;</th>
<th>fun() -&gt; 1+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>4</td>
</tr>
</tbody>
</table>

Client environment

Injections

"i" ⟷ "r"
Execution of the compiled code

**ELIOM program**

```elm
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

**Server program**

```elm
x ← "r"
```

**Queue**

```
let y = 3 + get "i"
return y
```

**Client program**

```
"i" ← "r"
```

**Injections**

```
"f0" ← fun() → 1+3
"r" ← 4
```

**Client environment**
Execution of the compiled code

ELIOM program

```
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

Server program

```
x ← "r"
```

Server environment

Queue

```
let y = 3 + "r"
return y
```

Client program

```
"i" ← "r"
"f0" ← fun()->1+3
"r" ← 4
```

Client environment
Execution of the compiled code

**ELIOM program**

```
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

**Server program**

- `x` is bound to "r"
- Queue
- Injections
- `"i"` is bound to "r"
- `"f0"` is bound to `fun() -> 1+3`
- `"r"` is bound to 4
- `y` is bound to 7

**Client program**

- `return y`
Execution of the compiled code

ELIOM program

```
let%server x = [%client 1 + 3]
let%client y = 3 + ~%x
return y
```

Server program

```
x ↦ "r"
```

Queue

```
"i" ↦ "r"
```

Client program

```
"f0" ↦ fun() => 1+3
"r" ↦ 4
y ↦ 7
```

Client environment

```
7
```

Result
Theorem (Compilation preserves semantics)

Given a slicable program $P$ which reduces to $v$ with a trace $\theta$. Then:

- The server compilation $\langle P \rangle_s$ reduces to the queue $\zeta$ and the injections $\zeta$ with the trace $\theta_s$.
- The client compilation $\langle P \rangle_c$, the queue $\zeta$ and the injections $\zeta$ reduces to the value $v$ with the trace $\theta_c$.
- $\theta$ is equal to the concatenation of $\theta_s$ and $\theta_c$. 
Theorem (Compilation preserves semantics)

If converters are well-behaved,
Given a slicable program \( P \) which reduces to \( v \) with a trace \( \theta \). Then:

- The server compilation \( \langle P \rangle_s \) reduces to the queue \( \xi \) and the injections \( \zeta \) with the trace \( \theta_s \).
- The client compilation \( \langle P \rangle_c \), the queue \( \xi \) and the injections \( \zeta \) reduces to the value \( v \) with the trace \( \theta_c \).
- \( \theta \) is equal to the concatenation of \( \theta_s \) and \( \theta_c \).
Type universes

Client and server types are distinct in ELIOM!

```ocaml
1 let%server s : int = 1 + 2
2 let%client c : int = ~%s + 1
```
Type universes

Client and server types are distinct in ELIOM!

```ocaml
1 let server s : int_s = 1 + 2
2 let client c : int_c = ~s + 1
```
How to typecheck injections?

- Client and server types are in distinct universes
- We send values from the server to the client

We need to specify how to send values! This problem is known as cross-stage persistency.

```ocaml
let%server s : int = 1 + 2
let%client c : int = cint%s + 1
```

With the predefined converters:

```ocaml
val%server cint : (int, int) converter
val%server cfrag : ('a fragment, 'a) converter
```
How to typecheck injections?

- Client and server types are in distinct universes
- We send values from the server to the client

We need to specify how to send values! This problem is known as cross-stage persistency.

```
1 let server s : int_s = 1 + 2
2 let client c : int_c = cint%s + 1
```

With the predefined converters:

```
1 val server cint : (int_s, int_c) converter
2 val server cfrag : ('a fragment, 'a) converter
```
Converters are “functions” that cross the client/server boundaries.

Definition

A converter is said “well-behaved” if it can be decomposed into a server serialization and a client deserialization function.

```haskell
type (\'a, \'b) converter = {
  serialize: \'a -> serial ;
  deserialize: (serial -> \'b) fragment ;
}
```
Semantics of converters

Converters are “functions” that cross the client/server boundaries.

Definition

A converter is said “well-behaved” if it can be decomposed into a server serialization and a client deserialization function.

```haskell
type server ('a, 'b[@client]) converter = {
    serialize: 'a -> serial ;
    deserialize: (serial -> 'b) fragment ;
}
```
Theorem (Compilation preserves typing)

Given a well typed program $P$, then the client and server compilation, $\langle P \rangle_s$ and $\langle P \rangle_c$ are also well typed. Types for the compiled programs can trivially be deduced from the original ones.

This theorem ensures that the ML parts of ELIOM programs are typed “like ML”.
1. Formalization
   - Semantics
   - Compilation

2. Type system

3. Module system
Why modules?

With the ELiOM language thus far, we have location-aware programming in expressions.

We also want location-aware programming in the large!

In particular, we want:

- A good integration with OCAML
- Ability to load libraries at a chosen location
- Signatures that inform us about locations
- Separate compilation

⇒ We need a module system that accounts for locations.
Why modules?

With the ELIOM language thus far, we have *location-aware* programming in expressions.

We also want *location-aware* programming in the large!

In particular, we want:
- A good integration with OCAML
- Ability to load libraries at a chosen location
- Signatures that inform us about locations
- Separate compilation

⇒ We need a module system that accounts for locations.
Why modules?

With the E\textsc{lion} language thus far, we have \textit{location-aware} programming in expressions.

We also want \textit{location-aware} programming in the large!

In particular, we want:

- A good integration with \textsc{OCaml}
- Ability to load libraries at a chosen location
- Signatures that inform us about locations
- Separate compilation

\Rightarrow We need a module system that accounts for locations.
Integration with OCAM

On top of client and server, there is also a third location, base, which is usable everywhere.

```ocaml
let%base f x = ...
let%client a = f 2
let%server b = f 5
```

Theorem (Base/ML correspondance)

ELIOM modules, expressions and types on base location correspond exactly to the ML language.

⇒ Compilation objects from the OCAM compiler can be reused directly!
Integration with OCAML

On top of **client** and **server**, there is also a third location, **base**, which is usable everywhere.

```ocaml
1 let%base f x = ...
2 let%client a = f 2
3 let%server b = f 5
```

Theorem (Base/ML correspondance)

**ELIOM** modules, expressions and types on base location correspond exactly to the ML language.

⇒ Compilation objects from the OCAML compiler can be reused directly!
Integration with OCAMLR

On top of **client** and **server**, there is also a third location, **base**, which is usable everywhere.

1. `let%base f x = ...`
2. `let%client a = f 2`
3. `let%server b = f 5`

Theorem (Base/ML correspondance)

**ELIOM** *modules, expressions and types on base location correspond exactly to the ML language.*

⇒ Compilation objects from the OCAMLR compiler can be reused directly!
Integration with OCAML

On top of **client** and **server**, there is also a third location, **base**, which is usable everywhere.

```ocaml
let f x = ... 
let%client a = f 2
let%server b = f 5
```

**Theorem (Base/ML correspondance)**

**ELIOM** *modules, expressions and types on base location correspond exactly to the ML language.*

⇒ Compilation objects from the OCAML compiler can be reused directly!
Modules and locations

We can also declare modules on the location of our choice! The content of the module must be the same than its location.

```ocaml
module client JsMap : sig
  type client 'a t

  val client empty : 'a t
  val client add : 'a t -> string -> 'a -> unit

  ...
end
```

We can even omit annotations inside the module!
We can also declare modules on the location of our choice! The content of the module must be the same than its location.

```ocaml
module client JsMap : sig
    type 'a t

    val empty : 'a t
    val add : 'a t -> string -> 'a -> unit

    ...

end
```

We can even omit annotations inside the module!
Mixed modules

We can also declare “mixed” modules which contain declarations in different locations.

```
module%mixed M = struct
  let f x = ...
  let%client c = f 2
  let%server s = f 5
end
```

You can then use the content of the module as expected:

```
let%client x = ... M.c ...
let%server y = ... M.s ...
```

But using them in the wrong location is prevented:

```
let%client x = ... M.s ... (*) Error! (*)
```
Mixed modules

We can also declare “mixed” modules which contain declarations in different locations.

```
module mixed M = struct
  let f x = ...
  let%client c = f 2
  let%server s = f 5
end
```

You can then use the content of the module as expected:

```
let%client x = ... M.c ...
let%server y = ... M.s ...
```

But using them in the wrong location is prevented:

```
let%client x = ... M.s ... (* × Error! *)
```
Mixed modules

We can also declare “mixed” modules which contain declarations in different locations.

```plaintext
module%mixed M = struct
  let f x = ...
  let%client c = f 2
  let%server s = f 5
end

You can then use the content of the module as expected:

```plaintext
let%client x = ... M.c ...

let%server y = ... M.s ...

But using them in the wrong location is prevented:

```plaintext
let%client x = ... M.s ... (* × Error! *)
```
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

\[ F(X) \]

We need a mechanism to modify locations in signatures.
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

$$F \ ( \ X \ )$$

<table>
<thead>
<tr>
<th>Functor location</th>
<th>Argument location</th>
<th>Result location</th>
</tr>
</thead>
<tbody>
<tr>
<td>base server</td>
<td>base server</td>
<td>base server</td>
</tr>
</tbody>
</table>

⇒ We need a mechanism to modify locations in signatures.
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

\[ F(\text{server}) \]

<table>
<thead>
<tr>
<th>Functor location</th>
<th>Argument location</th>
<th>Result location</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>base server</td>
<td>base server</td>
</tr>
<tr>
<td>server</td>
<td>base server</td>
<td>base server</td>
</tr>
</tbody>
</table>

⇒ We need a mechanism to modify locations in signatures.
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

\[
F(X)
\]

<table>
<thead>
<tr>
<th>Functor location</th>
<th>Argument location</th>
<th>Result location</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>base</td>
<td>base</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>base</td>
<td>server</td>
</tr>
</tbody>
</table>

⇒ We need a mechanism to modify locations in signatures.
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

\[ F(X) \]

- **Functor location**: base, server, server, base
- **Argument location**: base, server, base, server
- **Result location**: base, server, server, ?

⇒ We need a mechanism to modify locations in signatures.
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

\[ F(X) \]

<table>
<thead>
<tr>
<th>Functor location</th>
<th>Argument location</th>
<th>Result location</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>base</td>
<td>base</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>base</td>
<td>server</td>
</tr>
<tr>
<td>base</td>
<td>server</td>
<td>server</td>
</tr>
</tbody>
</table>

⇒ We need a mechanism to modify locations in signatures.
What about locations and Functors?

The location of the result of the functor depends on the location of the functor and its argument.

$$F\left(X\right)$$

<table>
<thead>
<tr>
<th>Functor location</th>
<th>Argument location</th>
<th>Result location</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>base</td>
<td>base</td>
</tr>
<tr>
<td>server</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>server</td>
<td>base</td>
<td>server</td>
</tr>
<tr>
<td>base</td>
<td>server</td>
<td>server</td>
</tr>
</tbody>
</table>

⇒ We need a mechanism to modify locations in signatures.
Polymorphism to the rescue

Consider this function application:

\[(f \ x)\]

We instantiate \(f\) to \(\text{int} \rightarrow \text{int}\) before typechecking the function application.

We can do something similar for locations and functors.
Specialization

Consider this function application:

\[ F(\_\_\_\_) \]

We “specialize” \( F \) to the current location before typechecking the functor application.
We only have one “location variable”: base
Consider this function application:

\[ F(X) \]

We “specialize” \( F \) to the current location before typechecking the functor application. We only have one “location variable”: base.
Specialization – details

functor\((M:S)T\) \rightarrow functor(\(M:[S])[T]\)
Mixed functors

We also have (limited) supports for mixed functors!

```ocaml
module type COMPARABLE = sig
  type t
  val compare : t -> t -> int
end

module mixed MixedMap (Key : COMPARABLE) = struct
  module M = Map.Make(Key)

  type server ('a, 'b) table = {
    srv : 'a M.t ;
    cli : 'b M.t fragment ;
  }

  let server add id v tbl = ...
end
```
Mixed functors vs. Specialization

Mixed functors are more difficult:

```ocaml
module type S = sig
  type t
end

module%mixed F (A : S) = struct
  type%server bilocated = {
    srv : A.t ;
    cli : A.t fragment ;
  }
end
```

- The body of a mixed functor can depend on a base declaration on both side.
  ⇒ Analogous to forall quantification in function arguments.
  ⇒ We can’t specialize the argument of a mixed functor!
Specialization – Mixed modules

```ocaml
1 sig
2  type%base t
3  val%client x : int
4  val%server y : t
5 end

functor_{mixed}(M:S)T  →  functor_{mixed}(M:S)[T]
```
Using mixed functors

Replicated Shared data-structures

```ocaml
module Cache (Key : T) = struct
  module M = Map.Make(Key)

  type shared ('a, 'b) table =
    ('a M.t, 'b M.t) Shared.t

  include client M

  let server add id v tbl =
    [%client M.add ~%id ~%v ~%tbl ];
    M.add id v.srv tbl.srv

  let server find id tbl =
    { svr = M.find id tbl ;
      cli = [%client M.find ~%id ~% tbl ] }

  (* ... *)
end
```
Conclusion

I presented my work on ELIOM, an extension of OCAML for tierless Web programming. During my thesis, I worked on:

- A formalization of ELIOM as an extension of OCAML.
  - Ensures correct communication
  - Slice tierless programs statically
  - Efficient execution

- New features:
  - A new typesystem featuring converters
  - A location-aware module systems

- A new implementation:
  - Compiler: https://github.com/ocsigen/ocaml-eliom
  - Runtime: https://github.com/ocsigen/eliomlang
Questions ?
Why functor and locations?

Imagine we want dictionaries where keys are JavaScript strings.

Application of a base functor to a client module

```ocaml
module%client JsString = struct
  type%client t = Js.string
  let%client compare = Js.compare_string
end

module%client JsMap = Map.Make(JsString)
```

Map.Make comes from the OCAML standard library, it’s on base!
4 Using converters: RPC

5 Implementation

6 Comparison

7 Bibliography
Remote Procedure Call (or RPC) is the action of a client calling the server *without loading a new page* and potentially getting a value back.
Remote Procedure Calls

A simplified RPC API:

```elio
rpc.eliomi

1 type server ('i,'o) t
2 type client ('i,'o) t = 'i -> 'o
3
4 val server create : ('i -> 'o) -> ('i, 'o) t
```

An example using Rpc
```elio
let server plus1 : ('i, 'o) Rpc.t = Rpc.create (fun x -> x + 1)
let client f x = ~server plus1 x + 1
```
Remote Procedure Calls

A simplified RPC API:

```eliomi
 rpc.eliomi
  type \texttt{server} ('i,'o) t 
  type \texttt{client} ('i,'o) t = 'i -> 'o 
  val \texttt{server} create : ('i -> 'o) -> ('i, 'o) t 
```

An example using \texttt{Rpc}

```eliomi
 let \texttt{server} plus1 : (int, int) Rpc.t = 
  Rpc.create (fun x -> x + 1) 
  let \texttt{client} f x = ~\texttt{plus1} x + 1 
```
Implementing RPC with converters

```ocaml

**type**%server ('i,'o) t = {
  url : string ;
  handler: 'i -> 'o ;
}

**type**%client ('i, 'o) t = 'i -> 'o

let%server serialize t = serialize_string t.url
let%client deserialize x =
  let url = deserialize_string x in
  fun i -> XmlHttpRequest.get url i

let conv = {
  serialize = serialize ;
  deserialize = [%client deserialize] ;
}

let%server create handler =
  let url = "/rpc/" ^ generate_new_id () in
  serve url handler ;
  { url ; handler }
```

We can now use **counter** and **Rpc** together!

```ocaml
let%server save_counter_rpc : (int, unit) Rpc.t = Rpc.create save_counter

let%server widget_with_save : Html.element =
  let f = [%client ~%save_counter_rpc] in
  counter f
```
Compilation

For each `.eliom` file:
- One `.cmi`
- Two `.cm[ox]`

We change the magic of `.cmis` that comes from `.eliom` files.

`cmi` lookup is more complicated:
- Two new options: `-client-I` and `-server-I`
- Practical hack: Special handling for `.client.cmi` and `.server.cmi` files.
Slicing

To track the current side:
- One global references (just like levels...)  
- Hacks to propagate sides inside exceptions (for error messages)

Slicing at the typedtree level
Manipulating typedtrees is very difficult, so we produce two parse trees, and retype client and server independently.
Internal representation

Prime directive of the implementation:

“Thou shall not change data structures”

- .cmi files are compatible. We only add extra attributes.
- Tooling works.
- We still change the magic number.

```ocaml
ident.ml

1 type t = { stamp: int; name: string; mutable flags: int }
2
3 let global_flag = 1
4 let predef_exn_flag = 2
5
6 let client_flag = 4
7 let server_flag = 8
```
An implementation for converters

A signature for converters

```ocaml
module type CONV = sig
  type%server t
  type%client t
  val%server serialize : t -> serial
  val%client deserialize : serial -> t
end

implicit%mixed String : CONV
  with type%server t = string and type%client t = string

implicit%mixed Fragment {M : sig type%client t end} : CONV
  with type%server t = M.t fragment
  and type%client t = M.t

val%client (~%) : {C : CONV} -> C.t(*server*) -> C.t(*client*)
```

- Uses modular implicits
- Leverage mixed functors
No static typing!
Tierless languages – UR/WEB

button.ur

```haskell
fun hint_button msg = 
  return <xml>
    <button onclick= {fn _ => alert msg} >
      Show hint
    </button>
  </xml>
```

button.urs

```haskell
val hint_button : string -> page
```

- Location information is not syntactic
- No separate compilation
Tierless languages – ELIOM

```eliom
let%server hint_button msg =
  button
    ~a:[a_onclick [%client fun _ -> alert ~%msg] ]
    [pcdata "Show hint"]
```

```eliom
val%server hint_button : string -> Html.element
```

- Static slicing during compilation
- Efficient execution
- Extension of OCAML, Part of the OCSSIGEN project
fun hint_button msg = 
  let val m = from server get msg in
  [<button onclick="[say alert m]">Show hint</button>
  </button>]

val hint_button : string -> html @ server

- Location directly inside the types.
- Support an arbitrary number of locations.
- No module system!
- No separate compilation!
ELIOM bibliography

Gabriel Radanne and Jérôme Vouillon and Vincent Balat
ELIOM: A core ML language for Tierless Web programming
https://hal.archives-ouvertes.fr/hal-01349774
APLAS 2016

Gabriel Radanne and Vasilis Papavasileiou and Jérôme Vouillon and Vincent Balat
ELIOM: tierless Web programming from the ground up
https://hal.archives-ouvertes.fr/hal-01407898
IFL 2016

Gabriel Radanne and Jérôme Vouillon
Tierless Modules
https://hal.archives-ouvertes.fr/hal-01485362