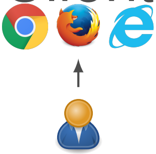
Tierless Web programming in ML

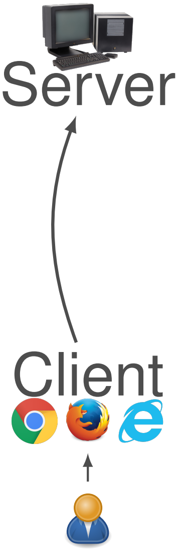
Gabriel RADANNE

Client



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

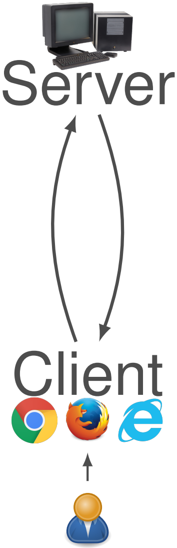
Bibliography
: Paper documentation on W3 and references.

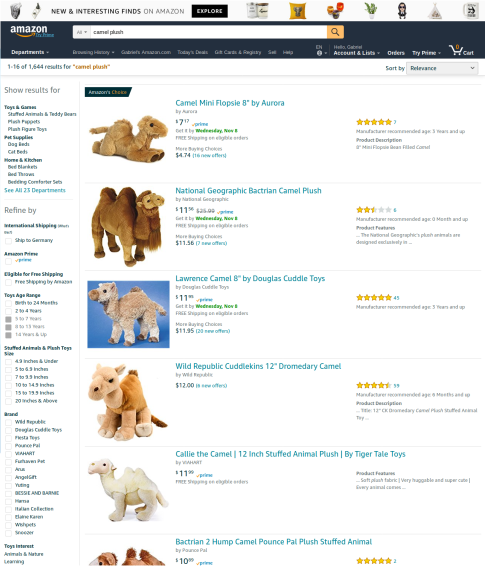
People
: A list of some people involved in the project.

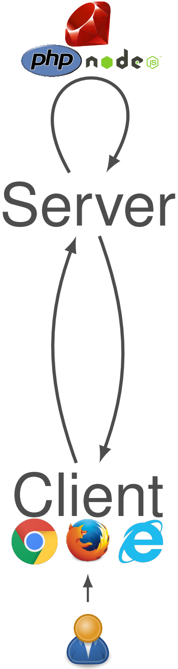
History
: A summary of the history of the project.

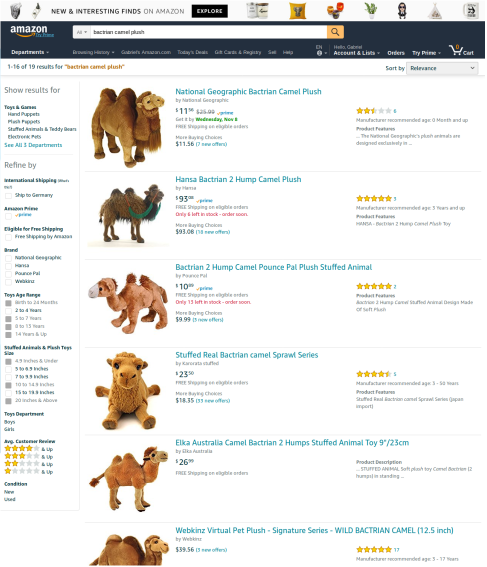
How can I help ?
: If you would like to support the web..

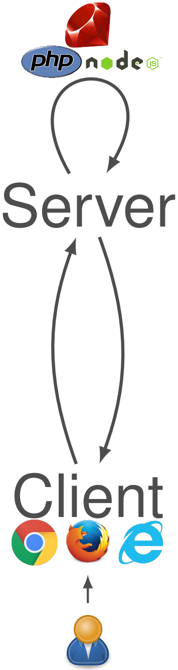
Getting code
: Getting the code by anonymous FTP , etc.

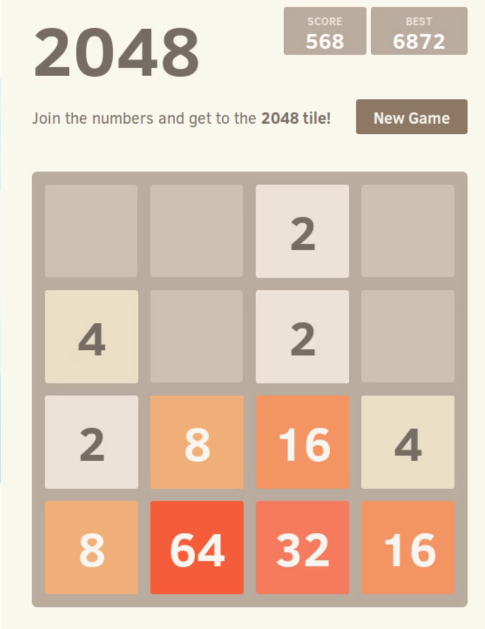


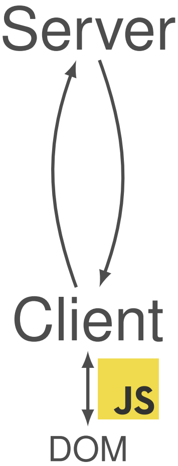

NEW & INTERESTING FINDS ON AMAZON
EXPLORE
amazon
Try Prime
All
camel plush
Departments
Browsing History
Gabriel's Amazon.com
Today's Deals
Gift Cards & Registry
Sell
Help
EN
Hello, Gabriel
Account & Lists
Orders
Try Prime
0
Cart
1-16 of 1,644 results for "camel plush"
Sort by
Relevance
Show results for
Toys & Games
Stuffed Animals & Teddy Bears
Plush Puppets
Plush Figure Toys
Pet Supplies
Dog Beds
Cat Beds
Home & Kitchen
Bed Blankets
Bed Throws
Bedding Comforter Sets
See All 23 Departments
Refine by
International Shipping
Ship to Germany
Amazon Prime
prime
Eligible for Free Shipping
Free Shipping by Amazon
Toys Age Range
Birth to 24 Months
2 to 4 Years
5 to 7 Years
8 to 13 Years
14 Years & Up
Stuffed Animals & Plush Toys Size
4.9 Inches & Under
5 to 6.9 Inches
7 to 9.9 Inches
10 to 14.9 Inches
15 to 19.9 Inches
20 Inches & Above
Brand
Wild Republic
Douglas Cuddle Toys
Fiesta Toys
Pounce Pal
VIAHART
Furhaven Pet
Arus
AngelGift
Yuting
BESSIE AND BARNIE
Hansa
Italian Collection
Elaine Karen
Wishpets
Snoozer
Toys Interest
Animals & Nature
Learning
Amazon's Choice
Camel Mini Flopsie 8" by Aurora
by Aurora
$7.17 prime
Get it by Wednesday, Nov 8
FREE Shipping on eligible orders
More Buying Choices
$4.74 (16 new offers)
7
Manufacturer recommended age: 3 Years and up
Product Description
8" Mini Flopsie Bean Filled Camel
National Geographic Bactrian Camel Plush
by National Geographic
$11.56 $25.99 prime
Get it by Wednesday, Nov 8
FREE Shipping on eligible orders
More Buying Choices
$11.56 (7 new offers)
6
Manufacturer recommended age: 0 Month and up
Product Features
... The National Geographic's plush animals are designed exclusively in ...
Lawrence Camel 8" by Douglas Cuddle Toys
by Douglas Cuddle Toys
$11.95 prime
Get it by Wednesday, Nov 8
FREE Shipping on eligible orders
More Buying Choices
$11.95 (20 new offers)
45
Manufacturer recommended age: 3 Years and up
Wild Republic Cuddlekins 12" Dromedary Camel
by Wild Republic
$12.00 (6 new offers)
59
Manufacturer recommended age: 6 Months and up
Product Description
... Title: 12" CK Dromedary Camel Plush Stuffed Animal Toy ...
Callie the Camel | 12 Inch Stuffed Animal Plush | By Tiger Tale Toys
by VIAHART
$11.99 prime
FREE Shipping on eligible orders
Product Features
... Soft plush fabric | Very huggable and super cute | Every animal comes ...
Bactrian 2 Hump Camel Pounce Pal Plush Stuffed Animal
by Pounce Pal
$10.89 prime
2

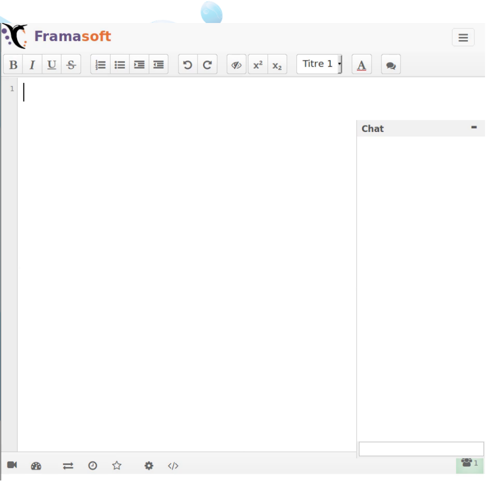

php
node
Server
Client

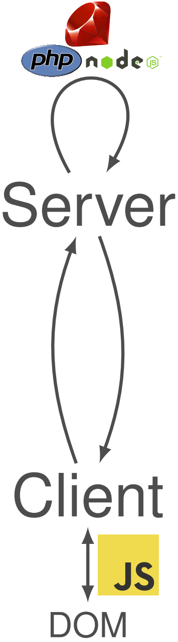

NEW & INTERESTING FINDS ON AMAZON
EXPLORE
amazon
Try Prime
All
bactrian camel plush
Departments
Browsing History
Gabriel's Amazon.com
Today's Deals
Gift Cards & Registry
Sell
Help
EN
Hello, Gabriel
Account & Lists
Orders
Try Prime
0
Cart
1-16 of 19 results for "bactrian camel plush"
Sort by
Relevance
Show results for
Toys & Games
Hand Puppets
Plush Puppets
Stuffed Animals & Teddy Bears
Electronic Pets
See All 3 Departments
Refine by
International Shipping
Ship to Germany
Amazon Prime
prime
Eligible for Free Shipping
Free Shipping by Amazon
Brand
National Geographic
Hansa
Pounce Pal
Webkinz
Toys Age Range
Birth to 24 Months
2 to 4 Years
5 to 7 Years
8 to 13 Years
14 Years & Up
Stuffed Animals & Plush Toys Size
4.9 Inches & Under
5 to 6.9 Inches
7 to 9.9 Inches
10 to 14.9 Inches
15 to 19.9 Inches
20 Inches & Above
Toys Department
Boys
Girls
Avg. Customer Review
& Up
& Up
& Up
& Up
Condition
New
Used
National Geographic Bactrian Camel Plush
by National Geographic
$11.56 $25.99 prime
Get it by Wednesday, Nov 8
FREE Shipping on eligible orders
More Buying Choices
$11.56 (7 new offers)
6
Manufacturer recommended age: 0 Month and up
Product Features
... The National Geographic's plush animals are designed exclusively in ...
Hansa Bactrian 2 Hump Camel Plush
by Hansa
$93.08 prime
FREE Shipping on eligible orders
Only 6 left in stock - order soon.
More Buying Choices
$93.08 (18 new offers)
3
Manufacturer recommended age: 3 Years and up
Product Features
HANSA - Bactrian 2 Hump Camel Plush Toy
Bactrian 2 Hump Camel Pounce Pal Plush Stuffed Animal
by Pounce Pal
$10.89 prime
FREE Shipping on eligible orders
Only 13 left in stock - order soon.
More Buying Choices
$9.99 (3 new offers)
2
Product Features
Bactrian 2 Hump Camel Stuffed Animal Design Made Of Soft Plush
Stuffed Real Bactrian camel Sprawl Series
by Karorata stuffed
$23.50
FREE Shipping on eligible orders
More Buying Choices
$18.35 (33 new offers)
5
Manufacturer recommended age: 3 - 50 Years
Product Features
Stuffed Real Bactrian camel Sprawl Series (japan import)
Elka Australia Camel Bactrian 2 Humps Stuffed Animal Toy 9"/23cm
by Elka Australia
$26.99
FREE Shipping on eligible orders
Product Description
... STUFFED ANIMAL Soft plush toy Camel Bactrian (2 humps) in standing ...
Webkinz Virtual Pet Plush - Signature Series - WILD BACTRIAN CAMEL (12.5 inch)
by Webkinz
$39.56 (3 new offers)
17
Manufacturer recommended age: 3 - 17 Years

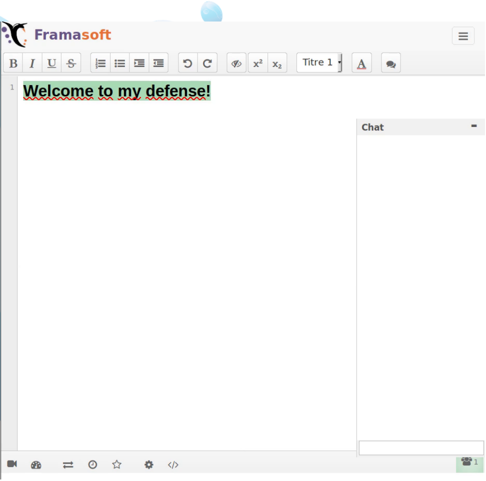

php
node
Server
Client

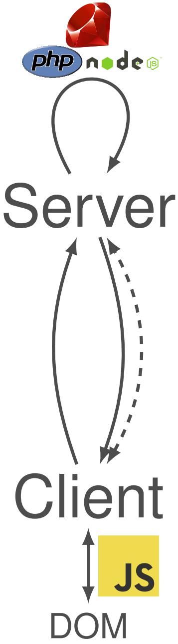

2048
SCORE
568
BEST
6872
Join the numbers and get to the **2048 tile!**
New Game

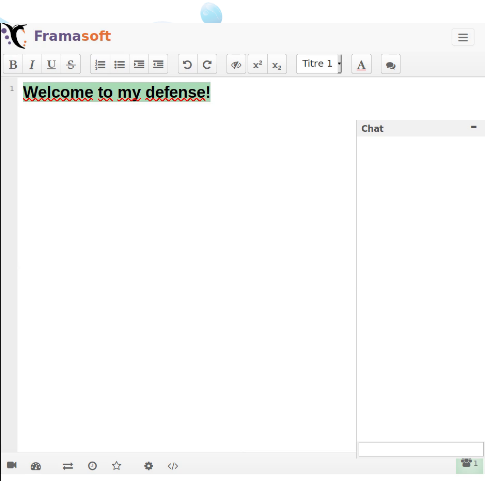

Server
Client
JS
DOM

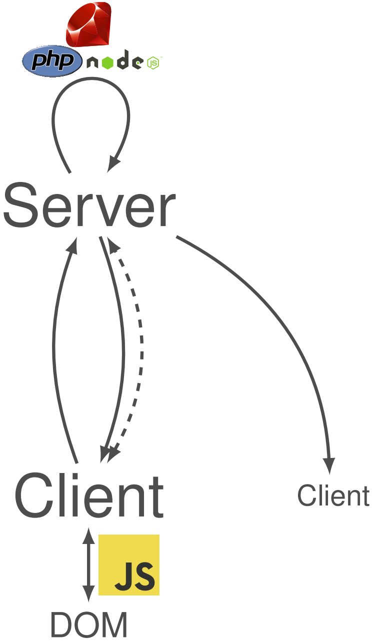

Framasoft
Titre 1
Chat

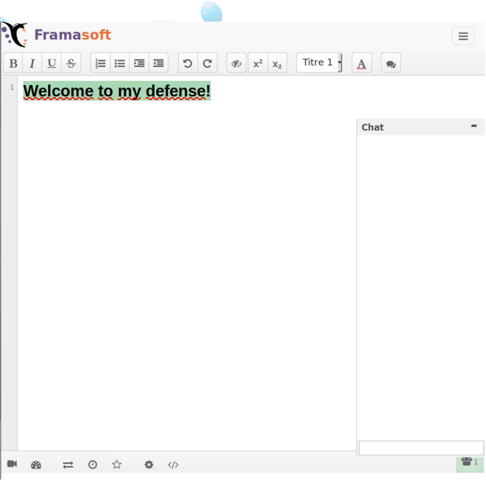

php
node
Server
Client
JS
DOM

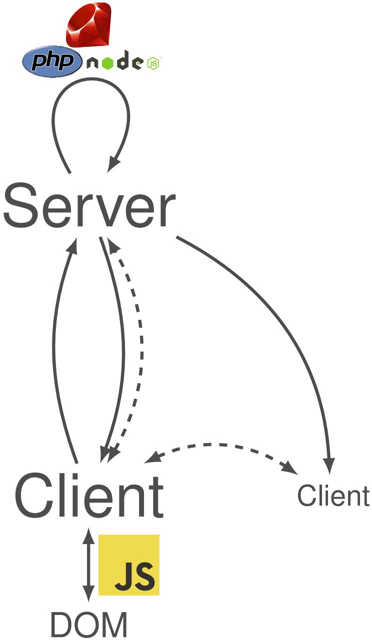

Framasoft
Titre 1
Welcome to my defense!
Chat

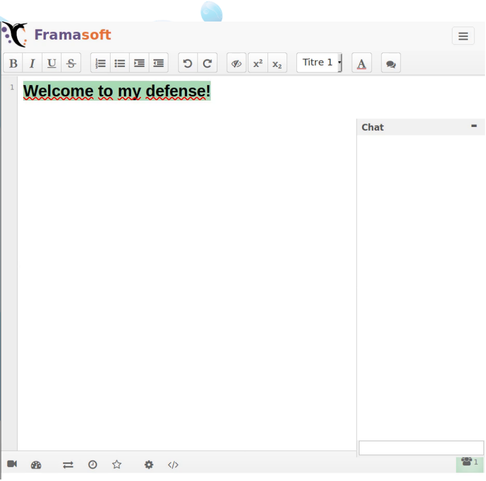

Server
Client
DOM

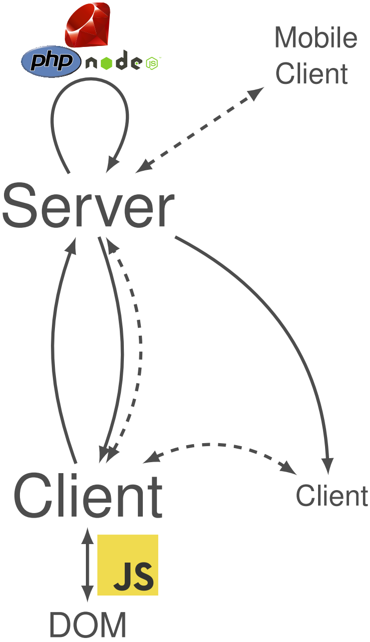

Framasoft
Titre 1
Welcome to my defense!
Chat


php
Server
Client
Client
JS
DOM

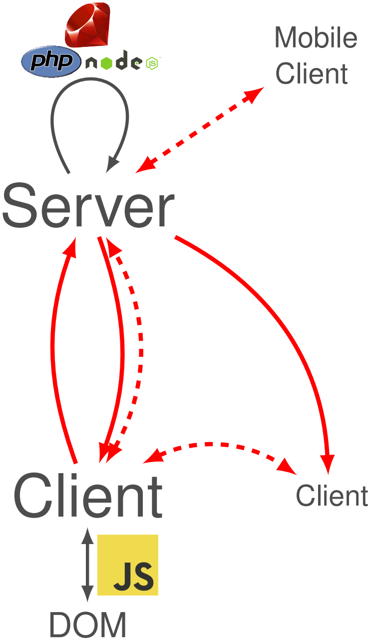

Framasoft
Titre 1
Welcome to my defense!
Chat


php
node
Server
Client
Client
JS
DOM


Framasoft
Titre 1
Welcome to my defense!
Chat


php
node JS
Mobile Client
Server
Client
Client
JS
DOM

Untyped

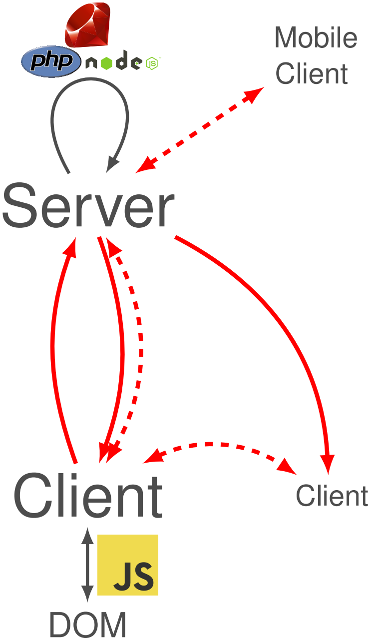


Server Send

line 1: Welcome to my defense!

Client Expect

line <number>: <text>

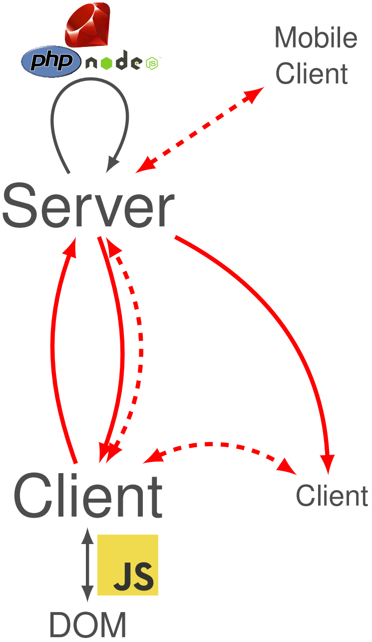


Server Send

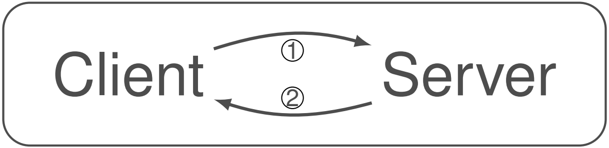
```
1,0:Welcome to my defense!
```

Client Expect

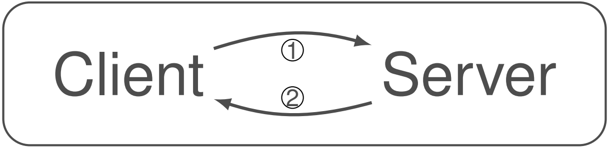
```
line <number>: <text>
```



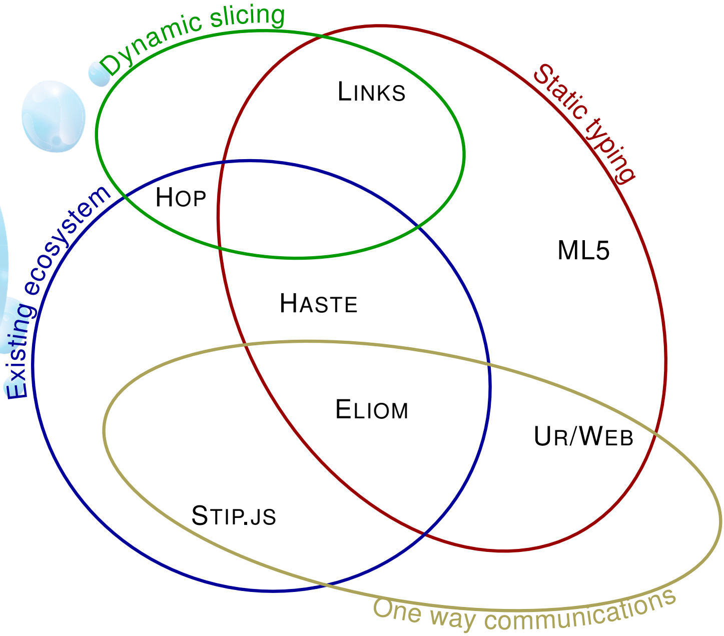
One program for everything

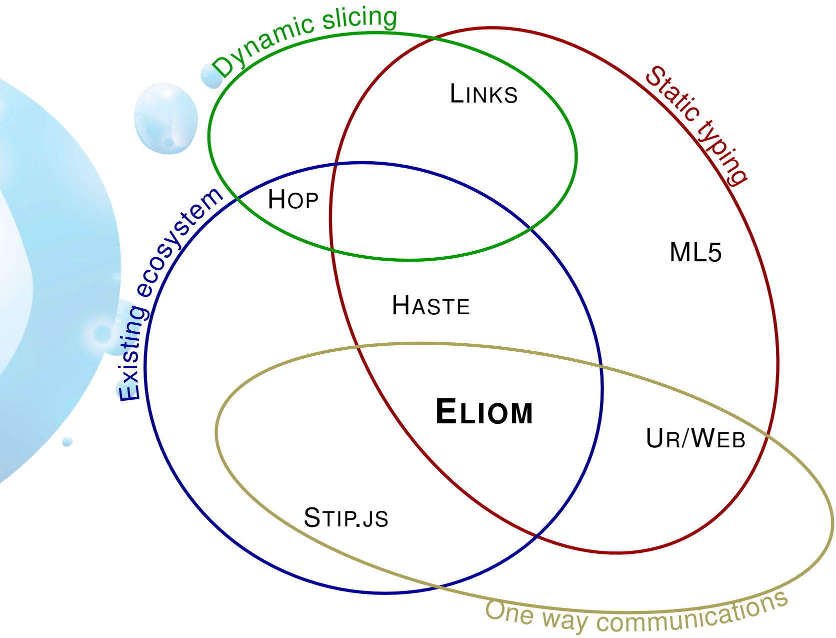


One program for everything



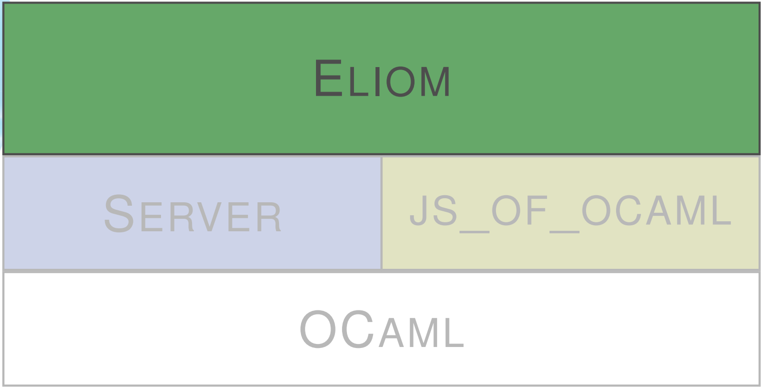
Tierless languages


Dynamic slicing
Static typing
Existing ecosystem
One way communications
LINKS
HOP
ML5
HASTE
ELIOM
UR/WEB
STIP.JS


Dynamic slicing
Static typing
Existing ecosystem
One way communications
LINKS
HOP
ML5
HASTE
ELIOM
UR/WEB
STIP.JS

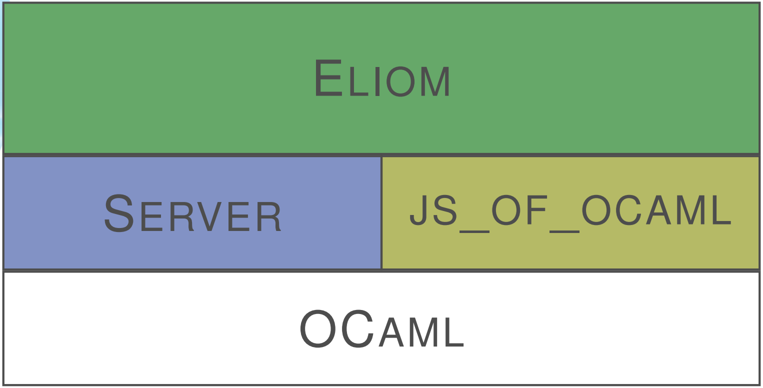
The OCSIGEN project

ocsigen
fresh air in web programming

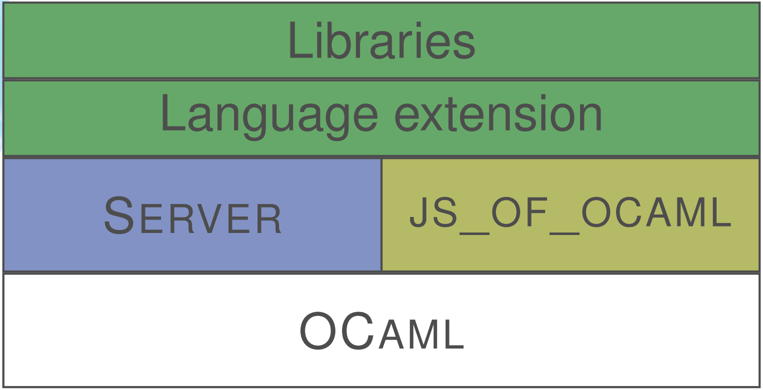


The OCSIGEN project

fresh air in web programming



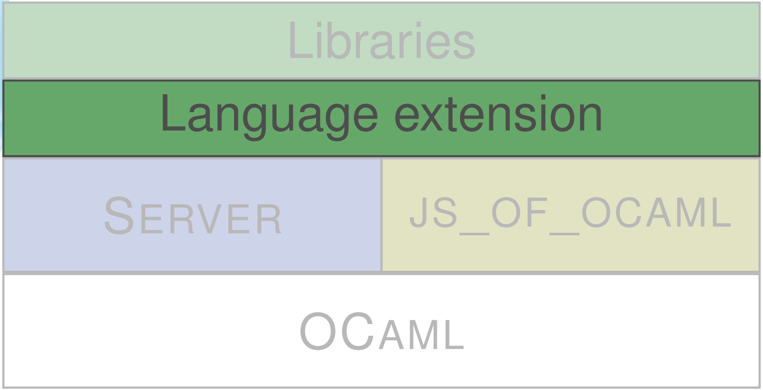
The OCSIGEN project



The OCSIGEN project

ocsigen

fresh air in web programming



Client and Server declarations

Client $\rightleftarrows$ Server

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

```
type%client t = ...

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.

```
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.

```
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.

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

Everything at once

We can combine injections and fragments.

```
let%server x : int fragment = [%client 1 + 3 ]
let%client c : int = 3 + ~%x
```

Small example – Hint button

button.eliom

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

button.html

```
<button onclick="...">
  Show hint
</button>
```

Small example – Hint button

button.eliom

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

button.html

```
<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 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.

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

```
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

```
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

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

ELIOM environment

Client program

Client environment

Example of execution

ELIOM program

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

ELIOM environment

Client program

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

Client environment

Example of execution

Eliom program

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

Eliom environment

Client program

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

Client environment

Example of execution

Eliom program

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

Eliom environment

$x \mapsto r$

Client program

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

Client environment

Example of execution

Eliom program

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

Eliom environment

$x \mapsto r$

Client program

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

Client environment

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
```

Client environment

Example of execution

ELIOM program

ELIOM environment

$x \mapsto r$

Client program

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

Client environment

Example of execution

ELIOM program

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

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

ELIOM environment

x ⟼ r

Client program

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

Client environment

f ⟼ **fun**()->1+3
r ⟼ 4

Example of execution

ELIOM program

ELIOM environment

x ↦ r

Client program

```
return y
```

Client environment

f ↦ **fun**()->1+3
r ↦ 4
y ↦ 7

Example of execution

ELIOM program

ELIOM environment

x ↦ r

Client program

Client environment

f ↦ `fun()->1+3`
r ↦ 4
y ↦ 7

Result

7

Example of compilation

Eliom code

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

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

Client code

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

Server code

Execution of the compiled code

ELIOM program

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

Server program

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

Server environment

Queue

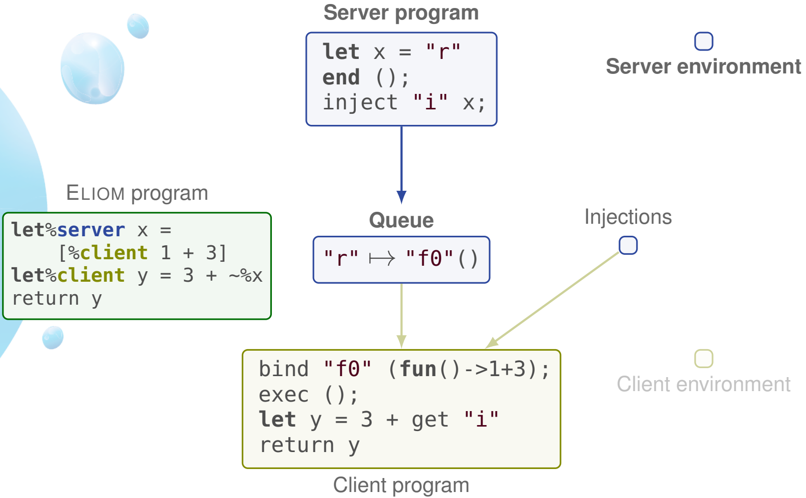
Injections

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

Client program

Client environment

Execution of the compiled code



Execution of the compiled code

Server program

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

```
x ↦ "r"
```

Server environment

ELIOM program

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

Queue

```
"r" ↦ "f0"()
```

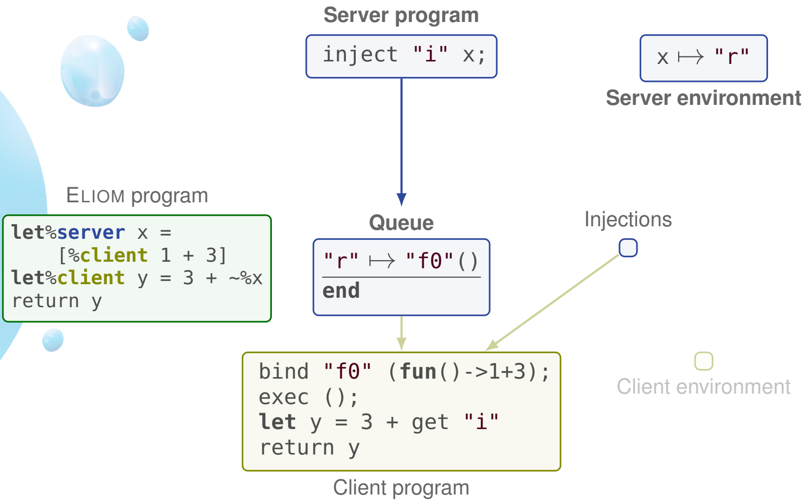
Injections

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

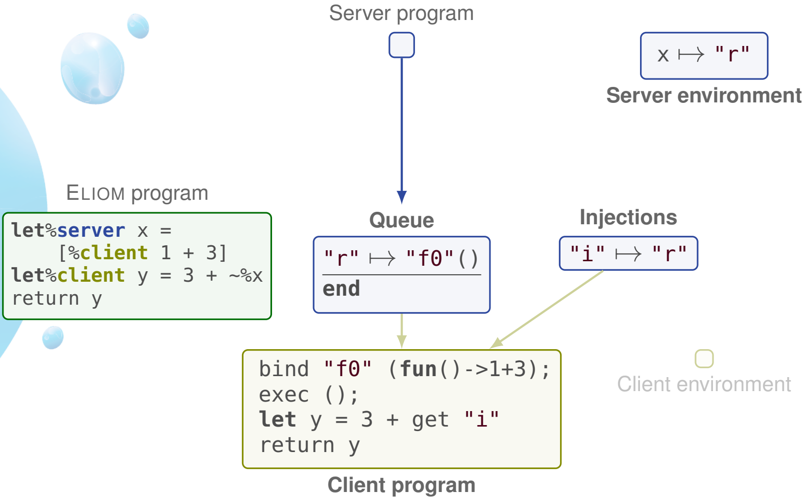
Client environment

Client program

Execution of the compiled code



Execution of the compiled code



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

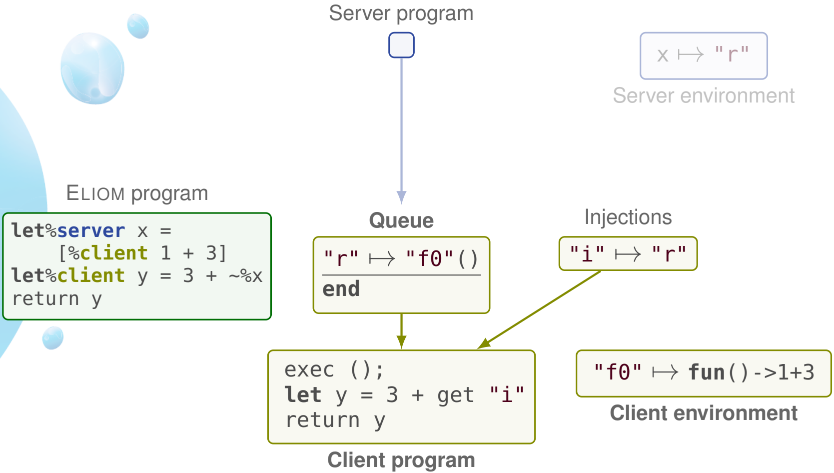
```
"i" ↦ "r"
```

Client environment

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

Client program

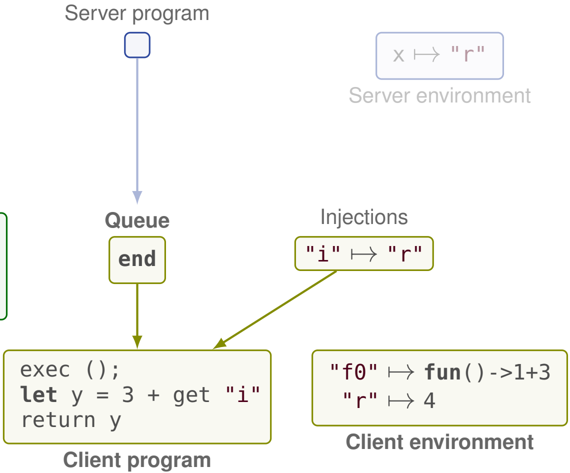
Execution of the compiled code



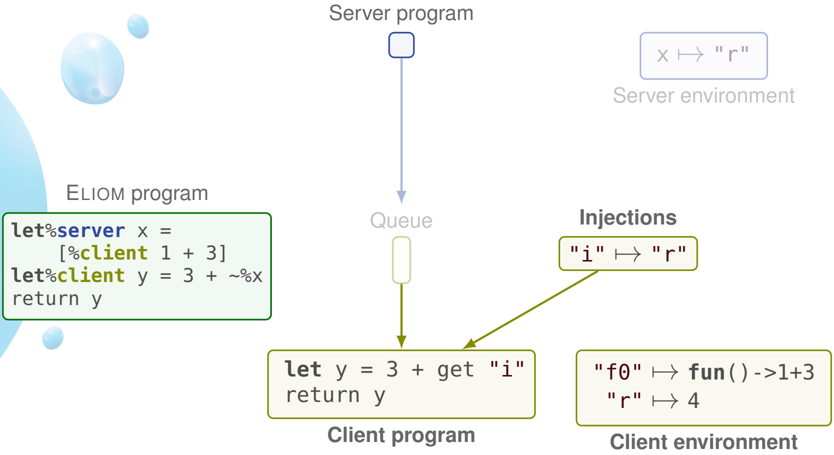
Execution of the compiled code

ELIOM program

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



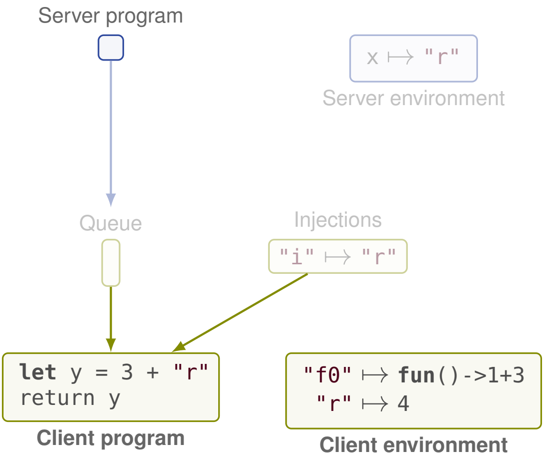
Execution of the compiled code



Execution of the compiled code

ELIOM program

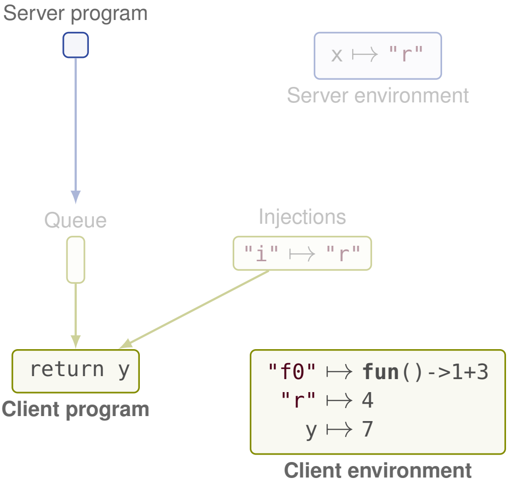
```
let%server x =
    [%client 1 + 3]
let%client y = 3 + ~%x
return y
```



Execution of the compiled code

ELIOM program

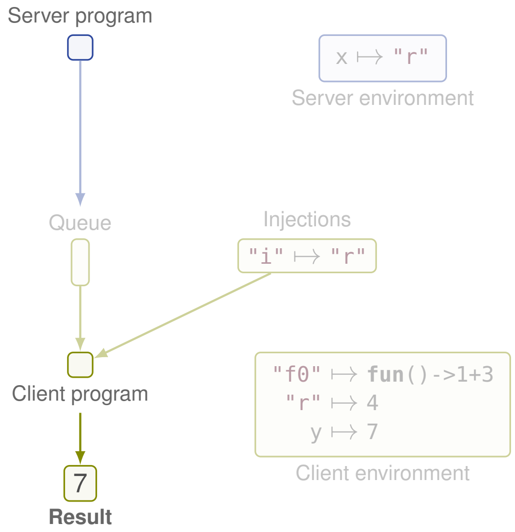
```
let%server x =
    [%client 1 + 3]
let%client y = 3 + ~%x
return y
```



Execution of the compiled code

ELIOM program

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



Theorem (Compilation preserves semantics)

Given a slicable program P which reduces to v with a trace θ. Then:

- *The server compilation $\langle P \rangle_s$ reduces to the queue ξ and the injections ζ with the trace θ_s.*
- *The client compilation $\langle P \rangle_c$, the queue ξ and the injections ζ reduces to the value v with the trace θ_c.*
- *θ is equal to the concatenation of θ_s and θ_c.*

Theorem (Compilation preserves semantics)

If converters are well-behaved,

Given a slicable program P which reduces to v with a trace θ. Then:

- *The server compilation $\langle P \rangle_s$ reduces to the queue ξ and the injections ζ with the trace θ_s.*
- *The client compilation $\langle P \rangle_c$, the queue ξ and the injections ζ reduces to the value v with the trace θ_c.*
- *θ is equal to the concatenation of θ_s and θ_c.*

Type universes

Client and server types are distinct in ELIOM!

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

Type universes

Client and server types are distinct in ELIOM!

```
let%server s : int_s = 1 + 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.

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

With the predefined converters:

```
val%server cint : (int_s, int_c) 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.

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

With the predefined converters:

```
val%server cint : (int_s, int_c) converter
val%server cfrag : ('a fragment, 'a) converter
```

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.

```
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.

```
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

$\Rightarrow$ 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 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.

Integration with OCaml

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

```
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 OCaml 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.

```
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 OCAML 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.

```
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.*

$\Rightarrow$ Compilation objects from the OCAML 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.

```
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.

```
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!

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.

```
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.

```
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! *)
```

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)

base ← → **base**

Functor location	Argument location	Result location
base	**base**	**base**

⇒ 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.

$$\underset{\text{server}}{F}\ (\ \underset{\text{server}}{X}\ )$$

Functor location	Argument location	Result location
base	**base**	**base**
server	**server**	**server**

$\Rightarrow$ 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.

$$F\ (\ X\ )$$

server **base**

Functor location	Argument location	Result location
base	**base**	**base**
server	**server**	**server**
server	**base**	?

⇒ 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)

server **base**

Functor location	Argument location	Result location
base	**base**	**base**
server	**server**	**server**
server	**base**	**server**

$\Rightarrow$ 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)

base ← F, X → **server**

Functor location	Argument location	Result location
base	**base**	**base**
server	**server**	**server**
server	**base**	**server**
base	**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.

$$\underset{\textbf{base}}{\text{F}} \; (\; \underset{\textbf{server}}{\text{X}} \;)$$

Functor location	Argument location	Result location
base	**base**	**base**
server	**server**	**server**
server	**base**	**server**
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)

base **server**

Functor location	Argument location	Result location
base	**base**	**base**
server	**server**	**server**
server	**base**	**server**
base	**server**	**server**

$\Rightarrow$ We need a mechanism to modify locations in signatures.

Polymorphism to the rescue

Consider this function application:

`( f x )`

$\alpha \to \alpha$ `int`

We instantiate *f* to `int` $\to$ `int` before typechecking the function application.

We can do something similar for locations and functors.

Specialization

Consider this function application:

$$F\ (\ X\ )$$

base **server**

We “specialize” *F* to the current location before typechecking the functor application.
We only have one “location variable”: base

Specialization

Consider this function application:

$$\underset{\textbf{server}}{F}\;(\;\underset{\textbf{server}}{X}\;)$$

We “specialize” *F* to the current location before typechecking the functor application.
We only have one “location variable”: base

Specialization – details

```
sig
  type%base t
  val%base x : t
end
```

$\longrightarrow$

```
sig
  type%client t
  val%client x : t
end
```

$$\texttt{functor}(M : S)\,T \longrightarrow \texttt{functor}(M : \lfloor S \rfloor)\,\lfloor T \rfloor$$

Mixed functors

We also have (limited) supports for mixed functors!

```
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:

```
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.

$\Rightarrow$ Analogous to forall quantification in function arguments.

$\Rightarrow$ We can't specialize the argument of a mixed functor!

Specialization – Mixed modules

```
sig
  type%base t
  val%client x : int
  val%server y : t
end
```

$\longrightarrow$

```
sig
  type%client t
  val%client x : int
end
```

$\texttt{functor}_{mixed}(M : S)T \longrightarrow \texttt{functor}_{mixed}(M : S)\lfloor T \rfloor$

Using mixed functors

Replicated Shared data-structures

```
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 =
    { srv = 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

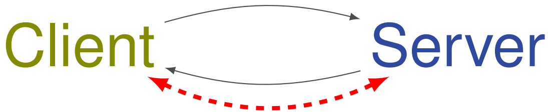
```
module%client JsString = struct
  type%client t = Js.string
  let%client compare = Js.compare_string
end

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

`Map.Make` comes from the OCAML standard library, it's on base!

Using converters for fun and profit

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:

rpc.eliomi

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

val%server create : ('i -> 'o) -> ('i, 'o) t
```

Remote Procedure Calls

A simplified RPC API:

rpc.eliomi

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

val%server create : ('i -> 'o) -> ('i, 'o) t
```

An example using Rpc

```
let%server plus1 : (int, int) Rpc.t =
  Rpc.create (fun x -> x + 1)

let%client f x = ~%plus1 x + 1
```

Implementing RPC with converters

```
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 }
```

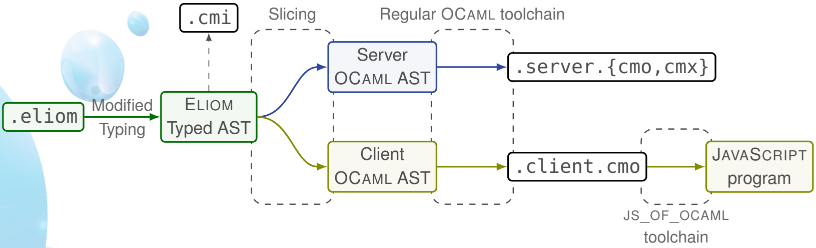
Widget + Rpc

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

```
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 `.cmi`s that comes from `.eliom` files.
- `cmi` lookup is a 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 parsetrees, 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.

ident.ml

```
type t = { stamp: int; name: string; mutable flags: int }

let global_flag = 1
let predef_exn_flag = 2

let client_flag = 4
let server_flag = 8
```

An implementation for converters

A signature for converters

```
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

Tierless languages – HOP

button.js

```
function hint_button (msg) {
  <button onclick= ~{alert (${msg}) } >
    Show hint
  </button>
}
```

No static typing!

Tierless languages – Ur/Web

button.ur

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

button.urs

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

- Location information is not syntactic
- No separate compilation

Tierless languages – ELIOM

button.eliom

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

button.eliomi

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

- Static slicing during compilation
- Efficient execution
- Extension of OCAML, Part of the OCSIGEN project

Tierless languages – ML5

button.ml5

```
fun hint_button msg =
  let val m = from server get msg in
  [<button onclick="[say alert m]">
     Show hint
   </button>]
```

button.mli5 – Not actually writable!

```
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