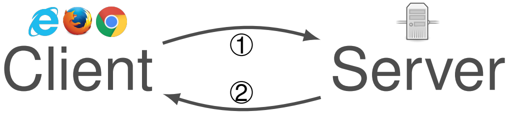
Eliom

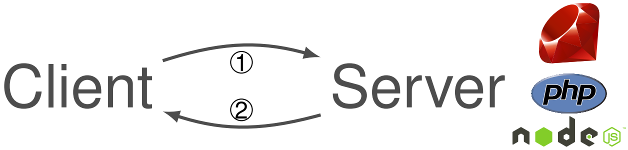
A core ML language for tierless Web programming

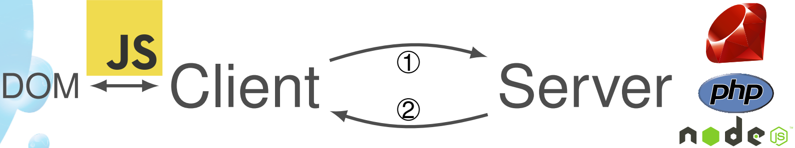
Gabriel Radanne

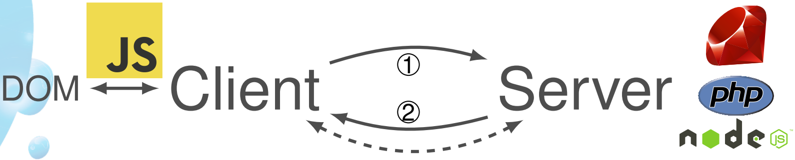
Jérôme Vouillon   Vincent Balat   Vasilis Papavasileiou

Evolution of the Web

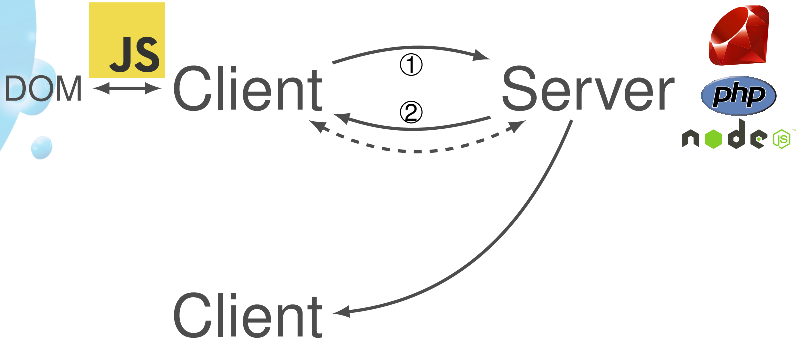


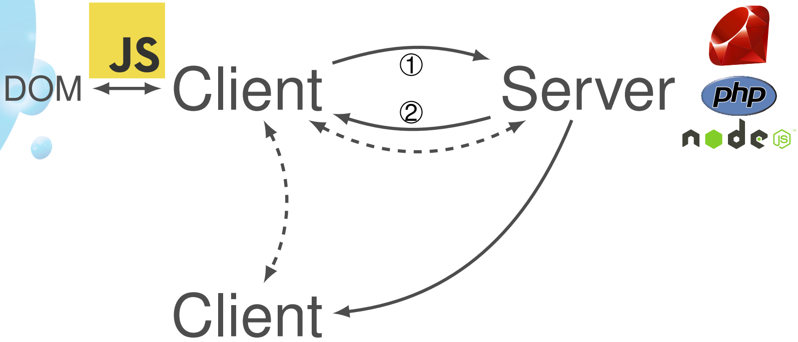

Client
①
②
Server
php
node JS

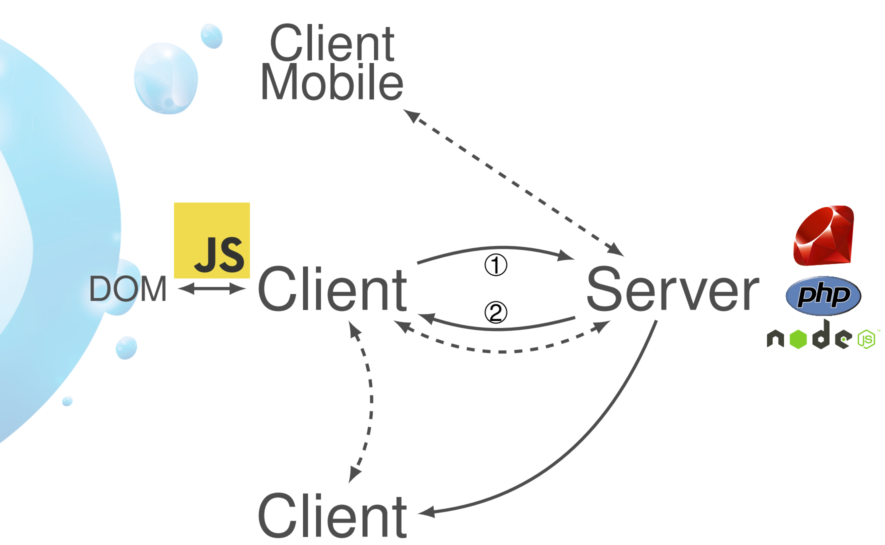

DOM
JS
Client
①
②
Server
php
node

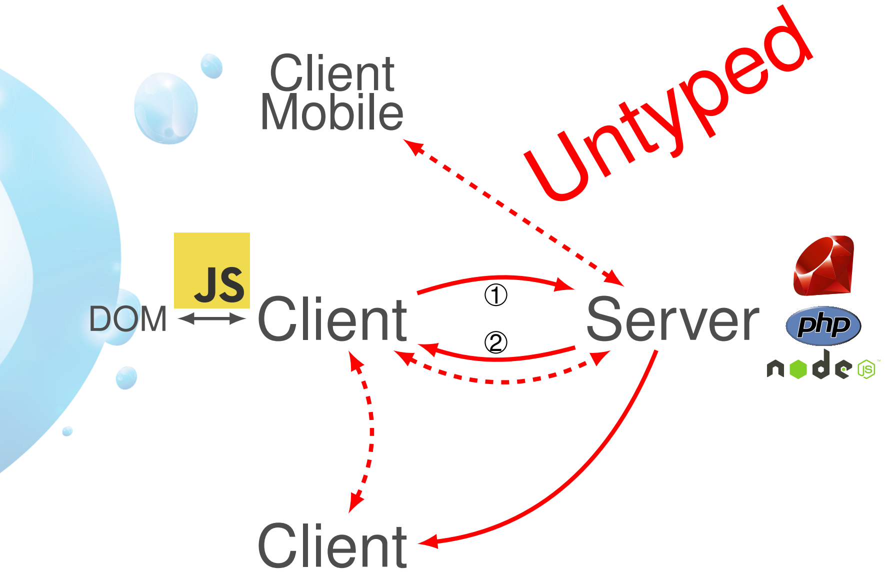


DOM
JS
Client
①
②
Server
php
node

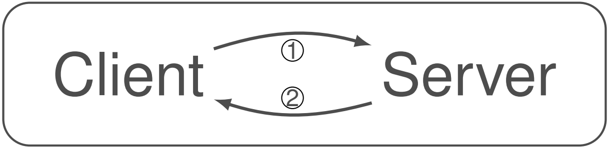

DOM
JS
Client
①
②
Server
php
node
Client


DOM
JS
Client
①
②
Server
php
node
Client

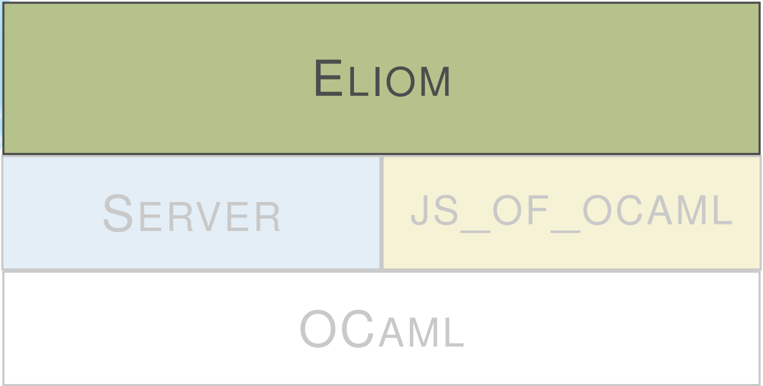

Client
Mobile
JS
DOM
Client
①
②
Server
php
node js
Client


Client
Mobile
Untyped
JS
DOM
Client
①
②
Server
php
node
Client

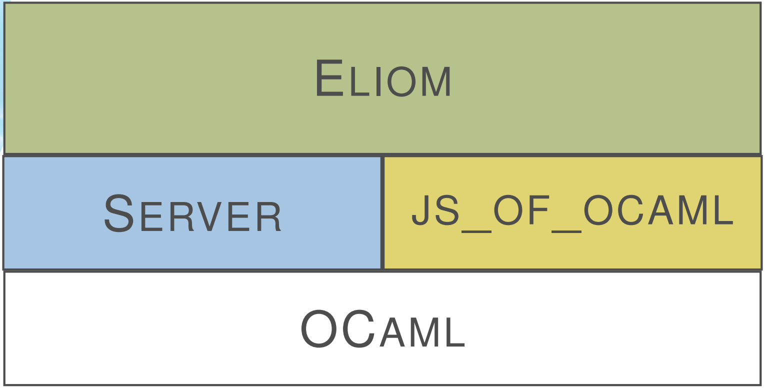
One program for everything



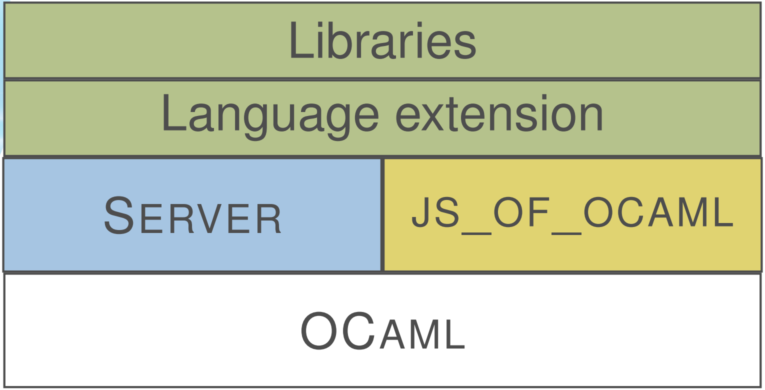
The OCSIGEN project



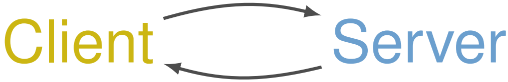
The OCSIGEN project



The OCSIGEN project



Client and Server annotations



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

```
let%server s = ...

let%client c = ...

let%shared sh = ...
```

The program is sliced during compilation.
This is important both for efficiency and predictability.

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
```

Counter widget

A button with a counter.

- HTML for the button is generated on the server.
- The button has a client-side state: the counter.
- When the button is pressed, the counter is incremented on the client.
- The button is parameterized by a client-side action.

Counter widget

counter.eliom

```
let%server counter action =
  let state = [%client ref 0 ] in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          [%client fun _ ->
              incr ~%state;
              ~%action !(~%state) ]]
    [pcdata "Increment"]
```

Counter widget

counter.eliom

```
let%server counter action =
  let state = [%client ref 0 ] in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          [%client fun _ ->
              incr ~%state;
              ~%action !(~%state) ]]
    [pcdata "Increment"]
```

counter.eliomi

```
val%server counter: (int -> unit) fragment -> Html.t
```

Counter widget

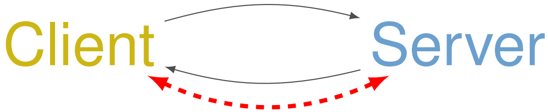
What if we want to save the state of the counter on the server ?

counter.eliomi

```
val%server counter: (int -> unit) fragment -> Html.t
```

Remote Procedure Calls

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
```

Converters

Converters are a way to *converts datatype between server and client*. Here is a schematized signature for ~%, the injection operator:

```
type%shared serial (* A serialization format *)

type%server ('a, 'b) converter = {
  serialize : 'a -> serial ;
  deserialize : (serial -> 'b) fragment ;
}

(* Not a real type signature *)
val%client (~%) :
  ('a, 'b) converter -> 'a (* server *) -> 'b (* client *)
```

Implementing RPC with converters

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

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

let serialize t = serialize_string t.url
let deserialize x =
  let url = deserialize_string x in
  fun i -> AJAX.get url i
```

Widget + Rpc

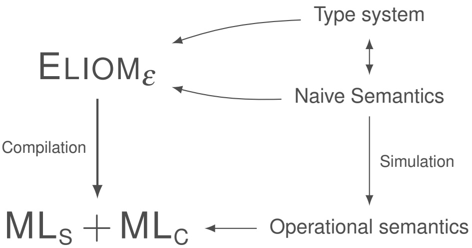
We can now use `counter` and `Rpc` together!

```
val%server save_counter : int -> unit
val%server counter : (int -> unit) fragment -> Html.t

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

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

The formalization



ELIOM$_\varepsilon$

Grammar:

$$p ::= \texttt{let}_s\ x = e_s\ \texttt{in}\ p \mid \texttt{let}_c\ x = e_c\ \texttt{in}\ p \mid e_c \qquad \text{(Programs)}$$

$$e_s ::= c_s \mid x \mid Y \mid (e_s\ e_s) \mid \lambda x.e_s \mid \{\{\ e_c\ \}\} \qquad \text{(Expressions)}$$

$$e_c ::= c_c \mid x \mid Y \mid (e_c\ e_c) \mid \lambda x.e_c \mid f\%e_s$$

$$f ::= x \mid c_s \qquad \text{(Converter)}$$

$$c_s \in Const_s \qquad c_c \in Const_c \qquad \text{(Constants)}$$

Types:

$$\sigma_\varsigma ::= \forall\alpha^*.\tau_\varsigma \qquad \text{(TypeSchemes)}$$

$$\tau_s ::= \alpha \mid \tau_s \to \tau_s \mid \{\tau_c\} \mid \tau_s \rightsquigarrow \tau_c \mid \kappa \text{ for } \kappa \in ConstType_s$$

$$\tau_c ::= \alpha \mid \tau_c \to \tau_c \mid \kappa \text{ for } \kappa \in ConstType_c \qquad \text{(Types)}$$

Meta-syntactic variables:

$$\varsigma \in \{c, s\}$$

Example

```
let%server s : int = 1 + 2

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

$$\texttt{let}_s\ s : \mathit{int}_s = 2\ \texttt{in}$$
$$\texttt{let}_c\ c : \mathit{int}_c = \texttt{cint\%}s + 1\ \texttt{in}$$
$$\ldots$$

Converters/Cross Stage Persistency

- 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!

$$\texttt{let}_s\ s : int_s = 2\ \texttt{in}$$
$$\texttt{let}_c\ c : int_c = \texttt{cint}\%s + 1\ \texttt{in}$$
$$\ldots$$

Given the converters:

$$\texttt{cint} : int_s \leadsto int_c$$
$$\texttt{fragment} : \forall\alpha.(\{\alpha\} \leadsto \alpha)$$

Converters/Cross Stage Persistency

- 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!

$$\texttt{let}_s\ s : int_s = 2\ \texttt{in}$$
$$\texttt{let}_c\ c : int_c = \texttt{cint}\%s + 1\ \texttt{in}$$
$$\ldots$$

Given the converters:

$$\texttt{cint} : int_s \rightsquigarrow int_c$$
$$\texttt{fragment} : \forall\alpha.(\{\alpha\} \rightsquigarrow \alpha)$$

Example with converters

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

let%client c : int = 3 + ~%x
```

$$\mathtt{let}_s\ x : \{\mathrm{int}_c\} = \{\{\ 1 + 3\ \}\}\ \mathtt{in}$$
$$\mathtt{let}_c\ y : \mathrm{int}_c = 3 + \mathtt{fragment\%}x\ \mathtt{in}$$
$$(y : \mathrm{int}_c)$$

Type system

Typing judgment: $(x_s : \sigma_s)_s, (x_c : \sigma_c)_c, \ldots \triangleright_\varsigma e : t$

$$\textsc{Var}\quad \frac{(x : \sigma)_\varsigma \in \Gamma \qquad \sigma \succ \tau}{\Gamma \triangleright_\varsigma x : \tau}$$

$$\textsc{Fragment}\quad \frac{\Gamma \triangleright_c e_c : \tau_c}{\Gamma \triangleright_s \{\{\ e_c\ \}\} : \{\tau_c\}}$$

$$\textsc{Injection}\quad \frac{\Gamma \triangleright_s f : \tau_s \rightsquigarrow \tau_c \qquad \Gamma \triangleright_s e_s : \tau_s}{\Gamma \triangleright_c f\%e_s : \tau_c}$$

One predefined constant types: `serial`
Two predefined converters:

$$\texttt{serial} : \texttt{serial} \rightsquigarrow \texttt{serial}$$
$$\texttt{fragment} : \forall\alpha.(\{\alpha\} \rightsquigarrow \alpha)$$

Example of execution

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1 + 3\ \}\}\ \texttt{in}$

$\texttt{let}_c\ y = 3 + \texttt{fragment}\%x\ \texttt{in}$

y

Queue

Example of execution

ELIOM code

$\texttt{let}_s\ x = \mathbf{r}\ \texttt{in}$

$\texttt{let}_c\ y = 3 + \texttt{fragment}\%x\ \texttt{in}$

y

Queue

$\mathbf{r} = 1 + 3$

Example of execution

ELIOM code

```
let_c y = 3 + fragment%r in
y
```

Queue

$\mathbf{r} = 1 + 3$

Example of execution

ELIOM code

$\texttt{let}_c\ y = 3 + \mathbf{r}\ \texttt{in}$
y

Queue

$\mathbf{r} = 1 + 3$

Example of execution

Eliom code

y

Queue

$\mathbf{r} = 1 + 3$
$y = 3 + \mathbf{r}$

Example of execution

ELIOM code

y

Queue

$\mathbf{r} = 4$
$y = 3 + \mathbf{r}$

Example of execution

ELIOM code

y

Queue

$y = 3 + 4$

Example of execution

Example of execution

Example of compilation

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1+3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$
y

Client code

$\texttt{bind}\ \mathbf{f_0} = \lambda().1+3\ \texttt{in}$
$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + \mathbf{i}\ \texttt{in}$
y

Server code

$\texttt{let}\ x = \texttt{fragment}\ \mathbf{f}_0\ ()\ \texttt{in}$
$\texttt{end}\ ();$
$\texttt{injection}\ \mathbf{i}\ x$

Execution of the compiled code

Server code

$\texttt{let}\ x = \texttt{fragment}\ \mathbf{f}_0\ ()\ \texttt{in}$
$\texttt{end}\ ();$
$\texttt{injection}\ \mathbf{i}\ x$

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1+3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment}\%x\ \texttt{in}$
y

Queue

Injections

$\texttt{bind}\ \mathbf{f_0} = \lambda().1+3\ \texttt{in}$
$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + \mathbf{i}\ \texttt{in}$
y

Client code

Execution of the compiled code

Server code

$\texttt{let}\ x = \mathbf{r}\ \texttt{in}$

$\texttt{end}\ ();$

$\texttt{injection}\ \mathbf{i}\ x$

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1+3\ \}\}\ \texttt{in}$

$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$

y

Queue

$\mathbf{r} = \mathbf{f}_0()$

Injections

$\texttt{bind}\ \mathbf{f_0} = \lambda().1 + 3\ \texttt{in}$

$\texttt{exec}\ ();$

$\texttt{let}\ y = 3 + \mathbf{i}\ \texttt{in}$

y

Client code

Execution of the compiled code

Server code

```
end ();
injection i r
```

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1+3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$
y

Queue

$\mathbf{r} = \mathbf{f}_0()$

Injections

$\texttt{bind}\ \mathbf{f_0} = \lambda().1+3\ \texttt{in}$
$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + \mathbf{i}\ \texttt{in}$
y

Client code

Execution of the compiled code

Server code

$\texttt{injection}\ \mathbf{i}\ \mathbf{r}$

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1 + 3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$
y

Queue

$\mathbf{r} = \mathbf{f}_0()$

$\texttt{end}$

Injections

$\texttt{bind}\ \mathbf{f_0} = \lambda().1 + 3\ \texttt{in}$
$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + \mathbf{i}\ \texttt{in}$
y

Client code

Execution of the compiled code

Server code

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1+3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment}\%x\ \texttt{in}$
y

Queue

$\mathbf{r} = \mathbf{f}_0()$
end

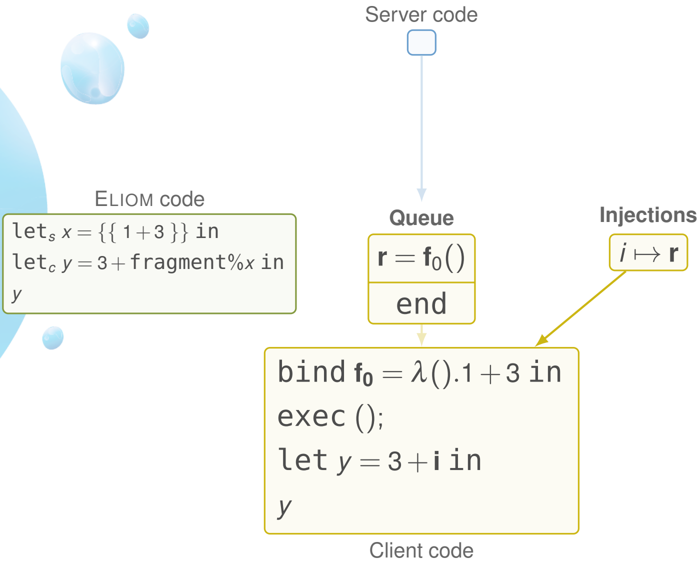
Injections

$i \mapsto \mathbf{r}$

$\texttt{bind}\ \mathbf{f_0} = \lambda().1+3\ \texttt{in}$
$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + \mathbf{i}\ \texttt{in}$
y

Client code

Execution of the compiled code



Execution of the compiled code

Server code

Eliom code

$\texttt{let}_s\ x = \{\{\ 1 + 3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$
y

Queue

$\mathbf{r} = \mathbf{f}_0()$

end

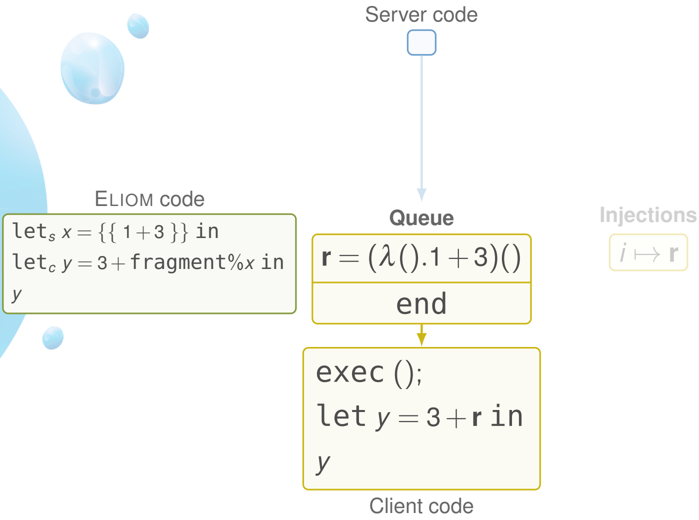
Injections

$i \mapsto \mathbf{r}$

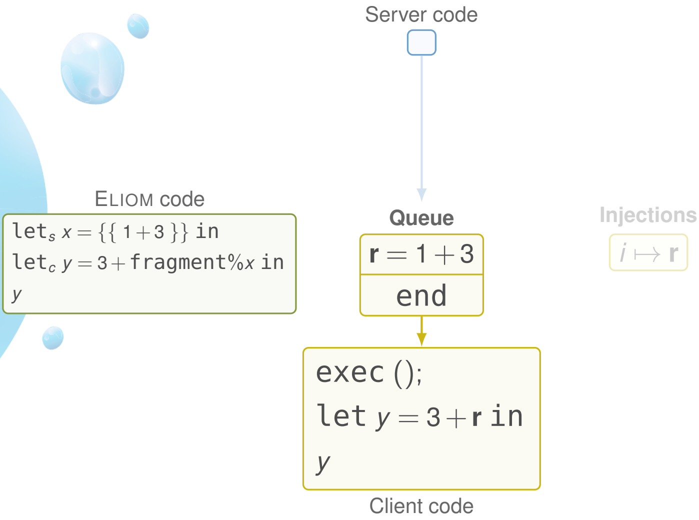
$\texttt{bind}\ \mathbf{f_0} = \lambda().1 + 3\ \texttt{in}$
$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + \mathbf{r}\ \texttt{in}$
y

Client code

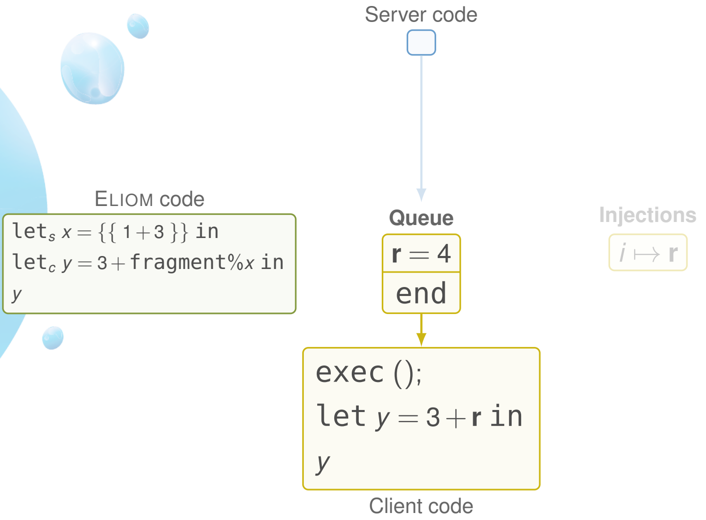
Execution of the compiled code



Execution of the compiled code



Execution of the compiled code



Execution of the compiled code

ELIOM code

$\texttt{let}_s\ x = \{\{\ 1 + 3\ \}\}\ \texttt{in}$
$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$
y

Server code

Queue

end

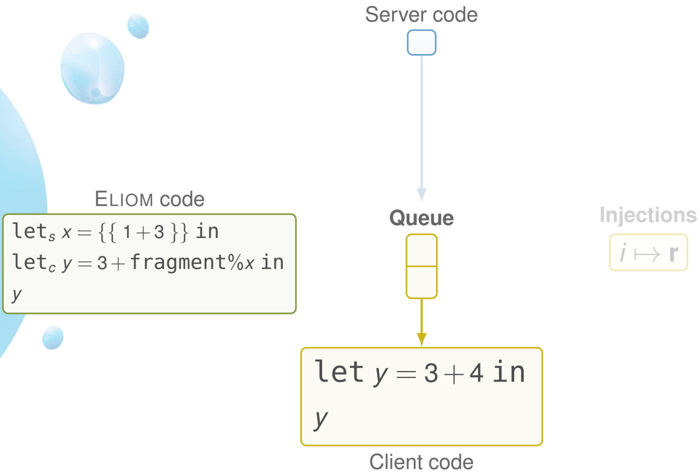
Injections

$i \mapsto \mathbf{r}$

$\texttt{exec}\ ();$
$\texttt{let}\ y = 3 + 4\ \texttt{in}$
y

Client code

Execution of the compiled code



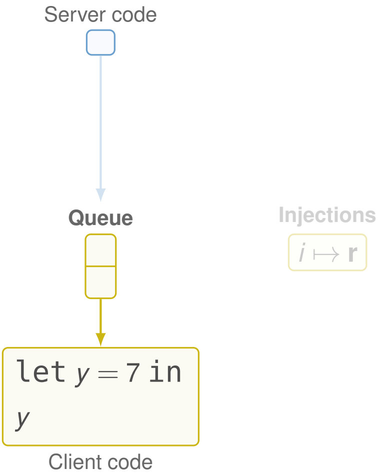
Execution of the compiled code

Eliom code

$\texttt{let}_s\ x = \{\{\ 1+3\ \}\}\ \texttt{in}$

$\texttt{let}_c\ y = 3 + \texttt{fragment\%}x\ \texttt{in}$

y



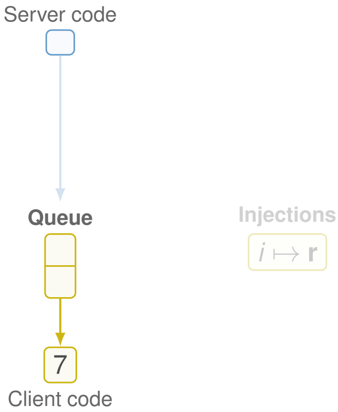
Execution of the compiled code

ELIOM code

$\mathtt{let}_s\ x = \{\{\ 1+3\ \}\}\ \mathtt{in}$

$\mathtt{let}_c\ y = 3 + \mathtt{fragment\%}x\ \mathtt{in}$

y



Subject Reduction

Theorem (Subject Reduction)

If $\xi_1 \vDash p_1 : \tau$ *and* $p_1 \mid \xi_1 \hookrightarrow p_2 \mid \xi_2$ *then* $\xi_2 \vDash p_2 : \tau$*.*

Simulation

Let ρ_s and ρ_c be the compilation functions from $\text{E}\textsc{liom}_\varepsilon$ to ML_S and ML_C.

Theorem (Simulation)

Let p be an $\text{E}\textsc{liom}_\varepsilon$ *program with an execution* $p \mid \emptyset \hookrightarrow^* v \mid \emptyset$

For an execution of p that terminates, we can exhibit a chained execution of $\rho_s(p)$ *and* $\rho_c(p)$ *such that evaluation is synchronized with p.*

Cross-side datatypes

```
type%server 'a fragment
```

'a is a *client* type.

Consider the piece of code:

```
type%server ('a, 'b) shared_frag = 'a * 'b fragment
```

How to preserve abstraction?

Cross-side datatypes

```
type%server 'a fragment
```

'a is a *client* type.

Consider the piece of code:

```
type%server ('a, 'b) shared_frag = 'a * 'b fragment
```

How to preserve abstraction?

Cross-side datatypes

foo.ml

```
type%server ('a, 'b[@client]) shared_frag = 'a * 'b fragment
```

foo.mli

```
type%server ('a, 'b[@client]) shared_frag
```

Module language

We have section annotations:

```
let%client x = ...
let%server x = ...
```

So many questions:

- Submodules ?
- What about signatures?
- Module subtyping?
- Regular OCAML modules?
- Functors?

Submodules

The typing rules and compilation scheme can be trivially extended to submodules:

```
module M = struct
  let%server x = ...
  let%client x = ...
end
```

We can have both x on the client and server.
What is the side of M ?
We need a notion of modules that are available on both sides.

Submodules

The typing rules and compilation scheme can be trivially extended to submodules:

```
module M = struct
  let%server x = ...
  let%client x = ...
end
```

We can have both `x` on the client and server.
What is the side of `M` ?
We need a notion of modules that are available on both sides.

Module subtyping

We need to check proper inclusion on sides:

```
module type S = sig
  val%client x : int
end

module M = struct
  let%client x = 3
  let%server x = 2
end

module%client C = M (* rejected: M is not client-only *)
module%client C = (M : S) (* accepted *)
```

We need to be careful with module aliases and the “lazyness” of the OCaml typechecker.

Module subtyping

We need to check proper inclusion on sides:

```
module type S = sig
  val%client x : int
end

module M = struct
  let%client x = 3
  let%server x = 2
end

module%client C = M (* rejected: M is not client-only *)
module%client C = (M : S) (* accepted *)
```

We need to be careful with module aliases and the “lazyness” of the OCaml typechecker.

Regular OCAML modules

We want to use regular OCAML modules on both sides.

```
let%client l = List.map ...
let%server x = String.length ...
```

Several solutions

- Load modules twice
- Add a new "base" side

Regular OCaml modules

We want to use regular OCaml modules on both sides.

```
let%client l = List.map ...
let%server x = String.length ...
```

Several solutions

- Load modules twice
- Add a new “base” side

Regular OCaml modules: Functors

We want to use regular OCaml functors on client/server modules:

```
module%client A = ...
module%client MapA = Map.Make(A)
```

Side polymorphism

ς was a meta-syntactic variable, we make it part of the grammar.

$$\ell ::= \varsigma \mid s \mid c$$

$$\textsc{VarPoly}\quad \frac{(x : \sigma)_\varsigma \in \Gamma \qquad \sigma[\varsigma/\ell] \succ \tau}{\Gamma \triangleright_\ell x : \tau}$$

We also specialize modules (by cutting out the part that is not on the current side).
=> Need to be very careful about strengthening.

- “Base” OCaml modules are appropriately specialized.
- “Mixed” modules can be projected to one side.
- There is only one side variable: No generalization problem (we are in a polymorphic context or we are not).

Side polymorphism

ς was a meta-syntactic variable, we make it part of the grammar.

$$\ell ::= \varsigma \mid s \mid c$$

$$\textsc{VarPoly}\quad \frac{(x : \sigma)_\varsigma \in \Gamma \qquad \sigma[\varsigma/\ell] \succ \tau}{\Gamma \triangleright_\ell x : \tau}$$

We also specialize modules (by cutting out the part that is not on the current side).
=> Need to be very careful about strengthening.

- "Base" OCaml modules are appropriately specialized.
- "Mixed" modules can be projected to one side.
- There is only one side variable: No generalization problem (we are in a polymorphic context or we are not).

Functors

We want to use modular implicits for converters:

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

We need functors containing side annotations:

- The type system works out fine
- The compilation scheme doesn't work at all. Either
  - Functor applications are synchronized:
    Easy, but useless for modular implicits.
  - Functor applications are not synchronized:
    We can't guarantee on the client that the server application happened.

We have ideas, but this is WIP.

Functors

We want to use modular implicits for converters:

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

We need functors containing side annotations:

- The type system works out fine
- The compilation scheme doesn't work at all. Either
  - Functor applications are synchronized:
    Easy, but useless for modular implicits.
  - Functor applications are not synchronized:
    We can't guarantee on the client that the server application happened.

We have ideas, but this is WIP.

The new implementation

Why ?

- As a mild chance of actually being sound (unlike the syntax extension)
- Much better inclusion in the OCaml language and error messages
- Can implement the new features (Module system, Mixed datatyped, . . .)

eliomlang

- A patch on the compiler (+2,874 -687 at the time of writing) `https://github.com/ocsigen/ocaml-eliom`
- Converters (as formalized) are not implementable right now.
- A new minimal runtime `https://github.com/ocsigen/eliomlang`
  - Barely any feature (only fragments and converters)
  - Only dependency is JS_OF_OCAML.

Some implementation details

We take two extra bits in `flags`:

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
```

By adding some attributes, we can keep the `cmi` format unchanged.

- 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.

Some implementation details – cont.

- OCAML's complicated compilation scheme cause issues.
  Interactions between separate compilation and ELIOM's dual type universe.
  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 lot more complicated:
  - Two new options: `-client-I` and `-server-I`
  - Practical hack: Special handling for `.client.cmi` and `.server.cmi` files.
- We want to also provide an ocamlbuild plugin to compile eliom projects easily…(So many nightmares)

Future work

Important tasks:

- Bring `eliomlang` to the point where we can use it for ELIOM
- Figure out functors, which would allow to implement converters.

Other tasks:

- An efficient serialization technique that works with converters.
- Extend the formalization to datatypes and modules.
- Formalize $\text{ELIOM}_{\varepsilon}$ in Coq (Started, got stuck).

Questions ?