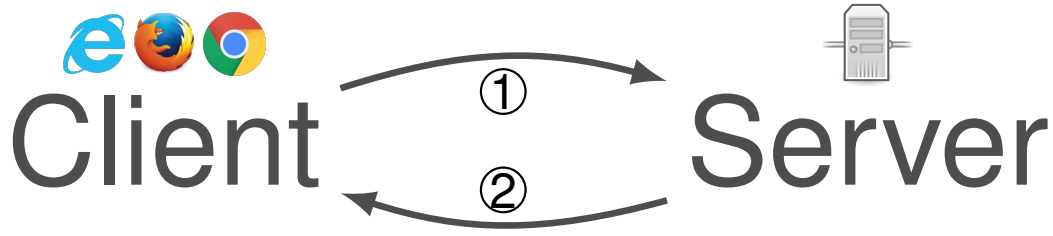
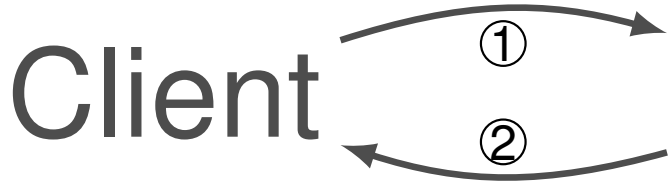
ELIOM

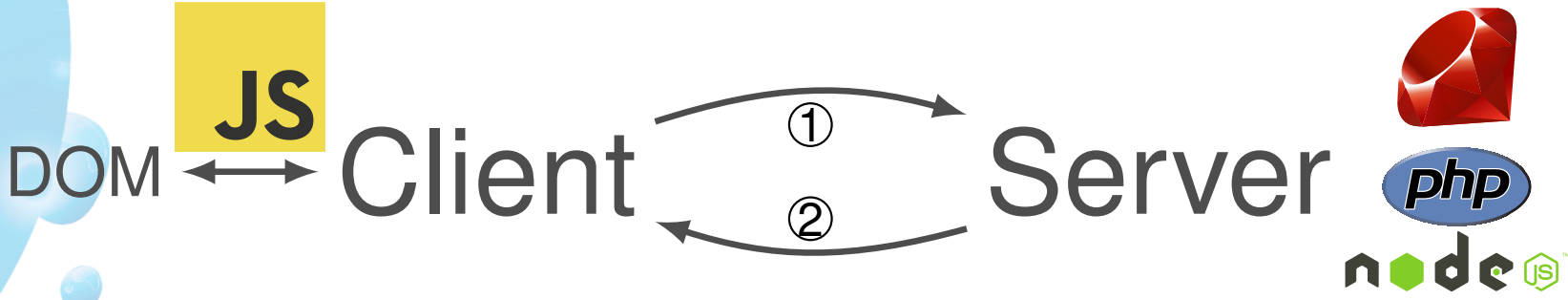
A core ML language for tierless Web programming

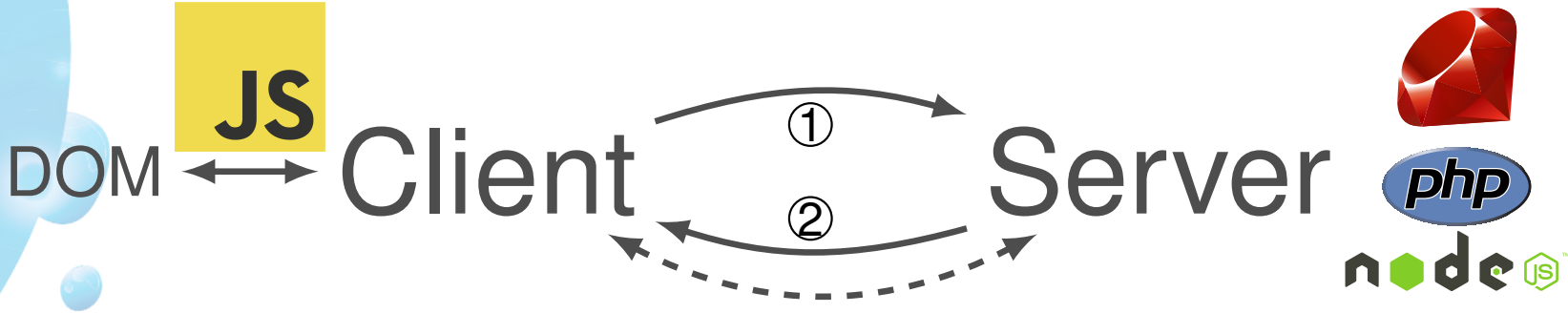
Gabriel RADANNE Jérôme VOUILLON Vincent BALAT

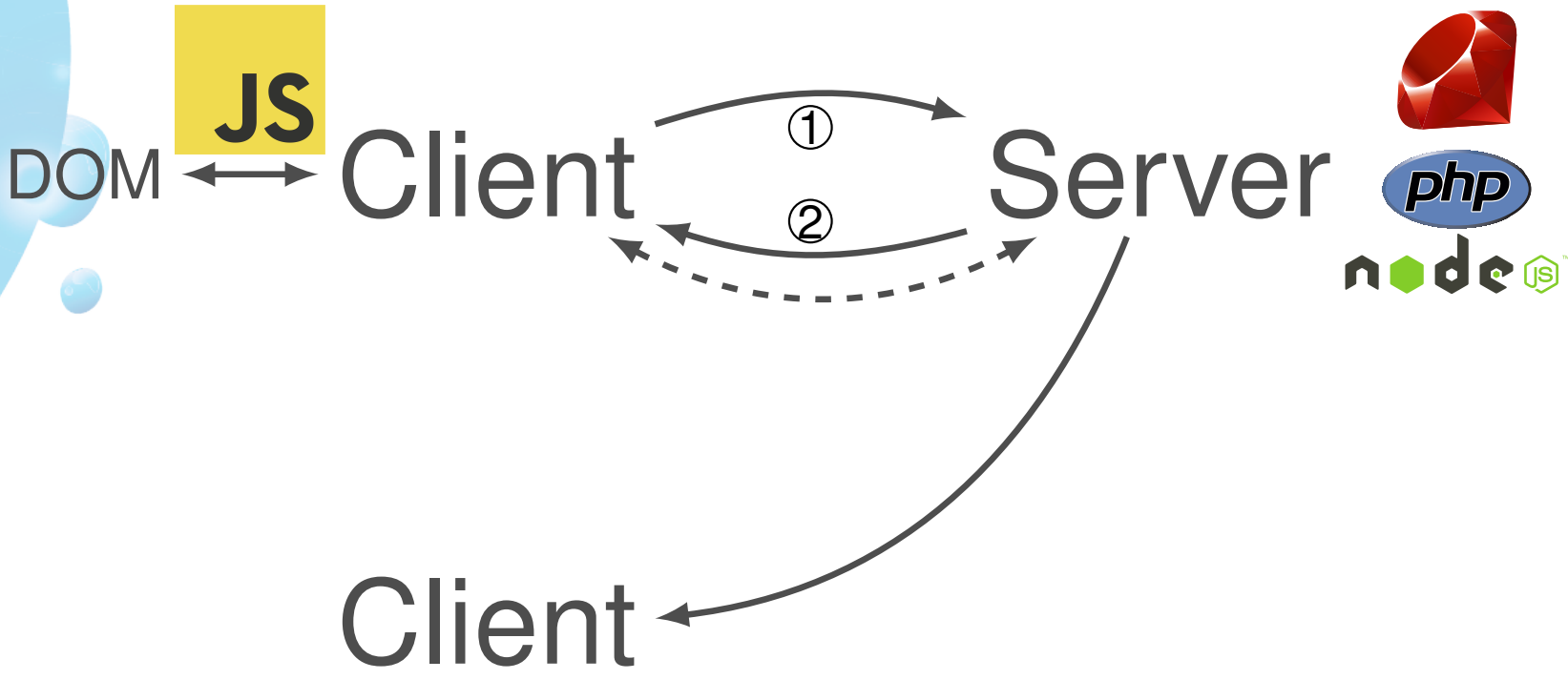
Evolution of the Web

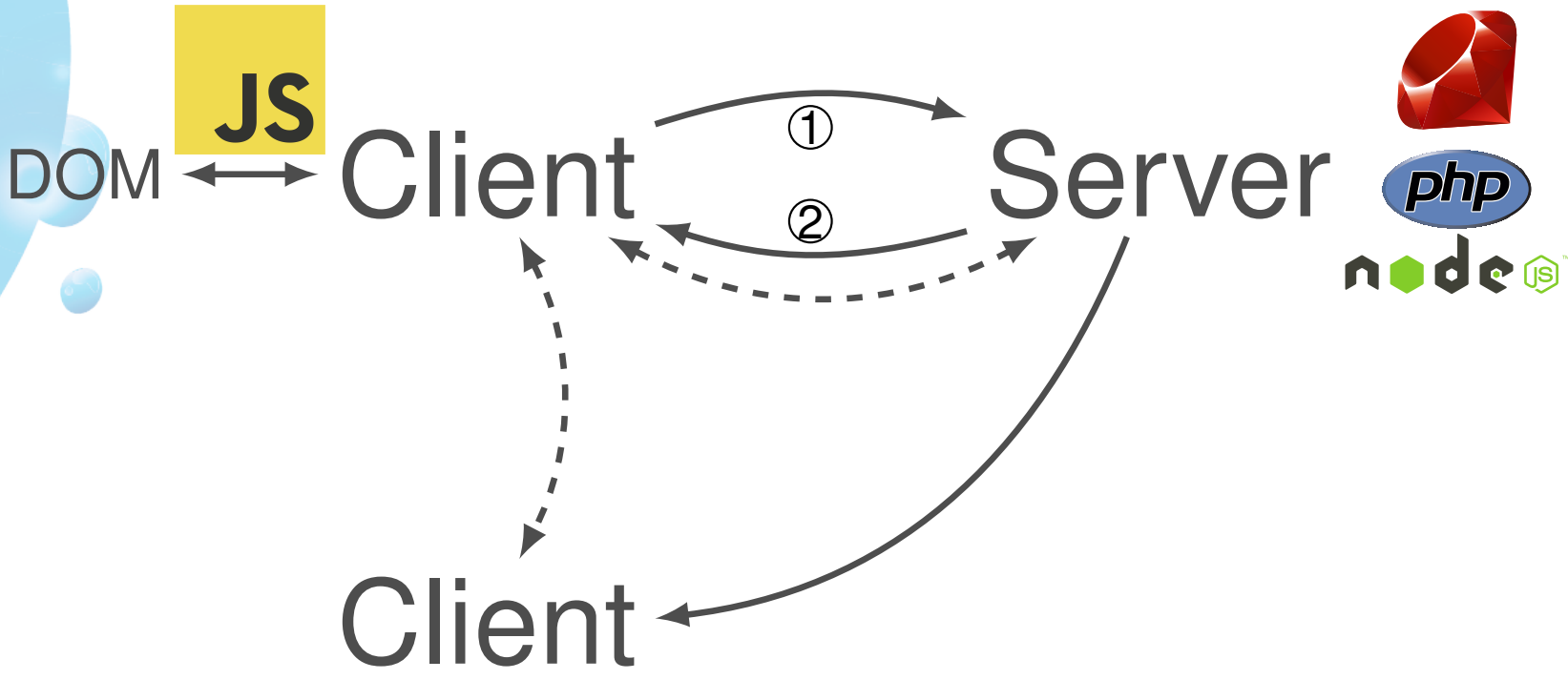


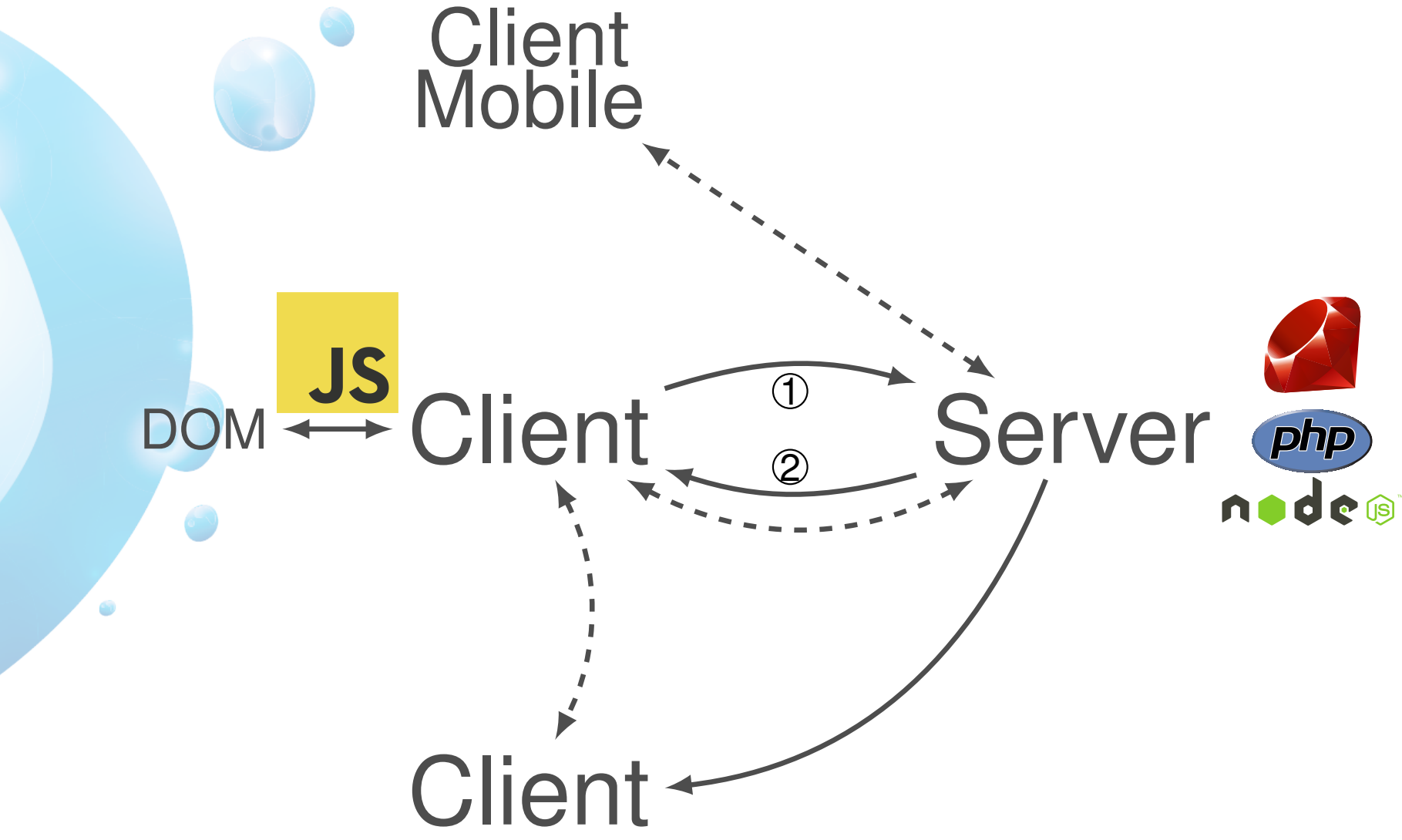

Client
①
②
Server

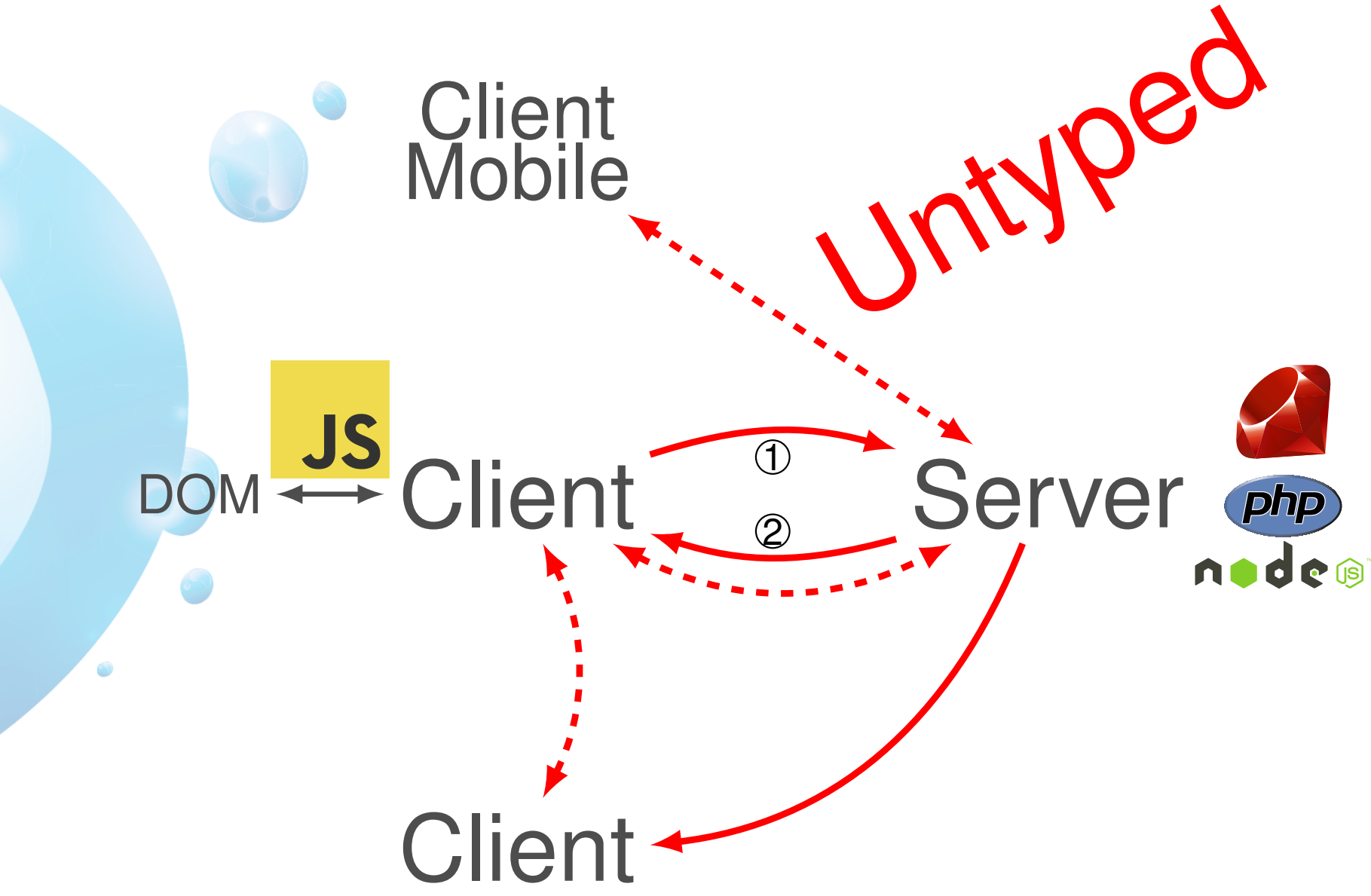

JS
DOM
Client
①
②
Server
php
node


JS
DOM
Client
①
②
Server
php
node

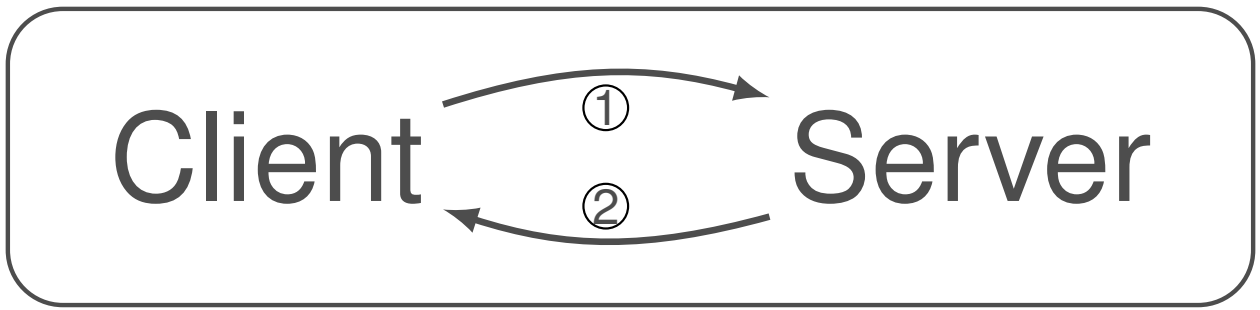

JS
DOM
Client
①
②
Server
Client


JS
DOM
Client
①
②
Server
php
node
Client

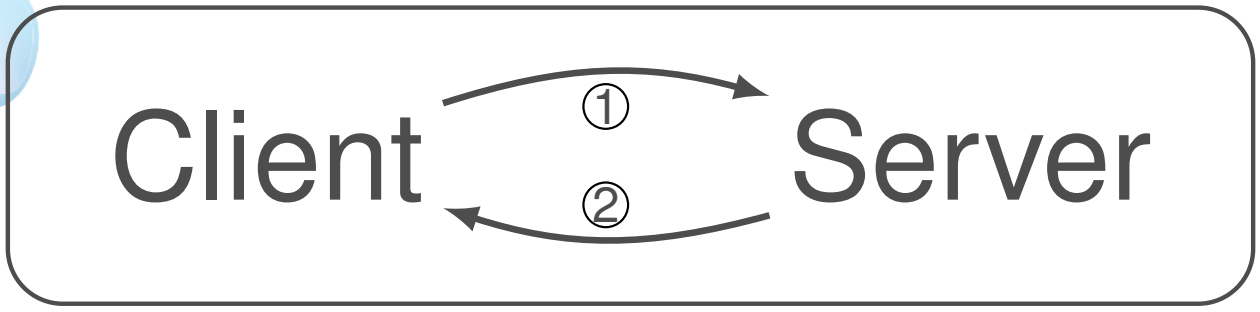

Client
Mobile
JS
DOM
Client
①
②
Server
php
node
Client


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

One program for everything



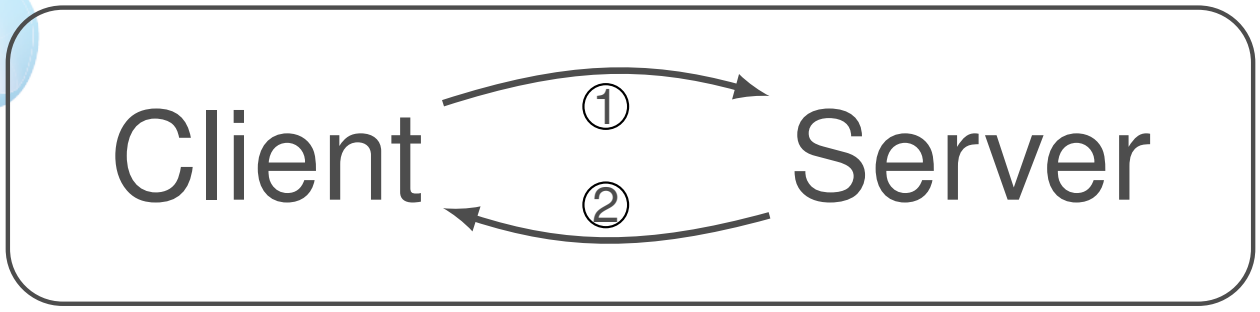
One program for everything



Tierless languages:

- LINKS
- HOP
- UR/WEB
- ELIOM

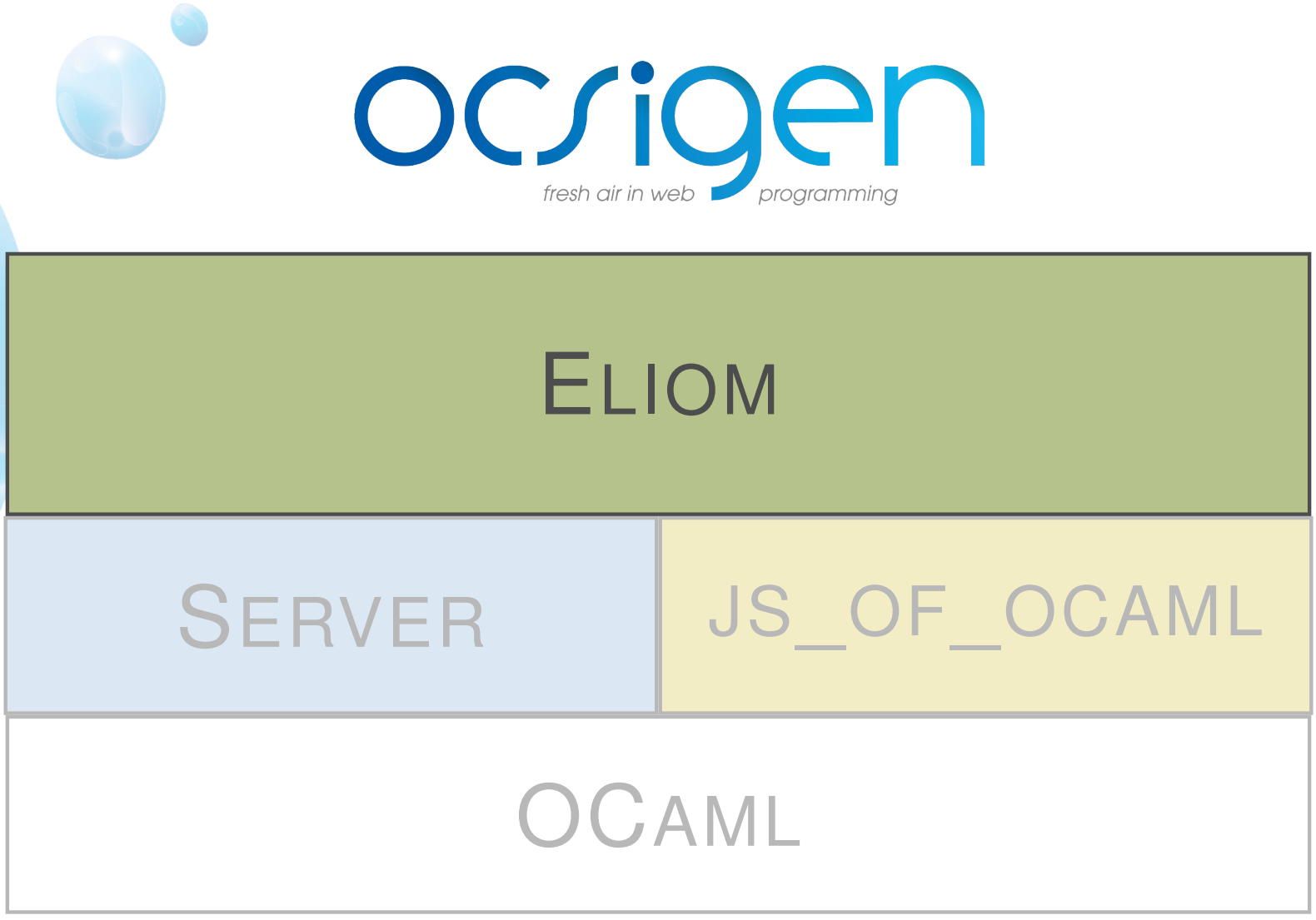
One program for everything



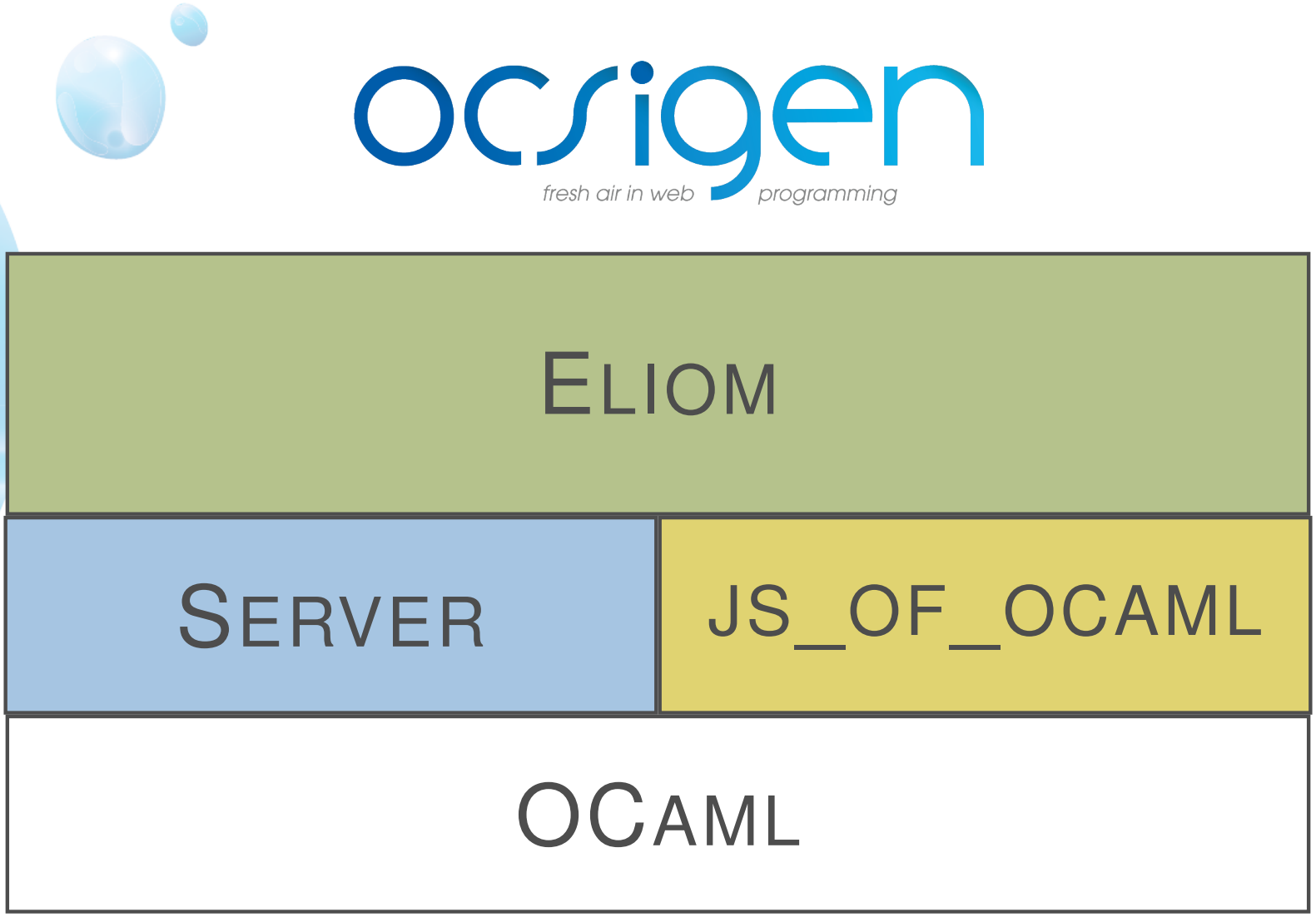
Tierless languages:

- LINKS
- HOP
- UR/WEB
- **ELIOM**

The OCSIGEN project



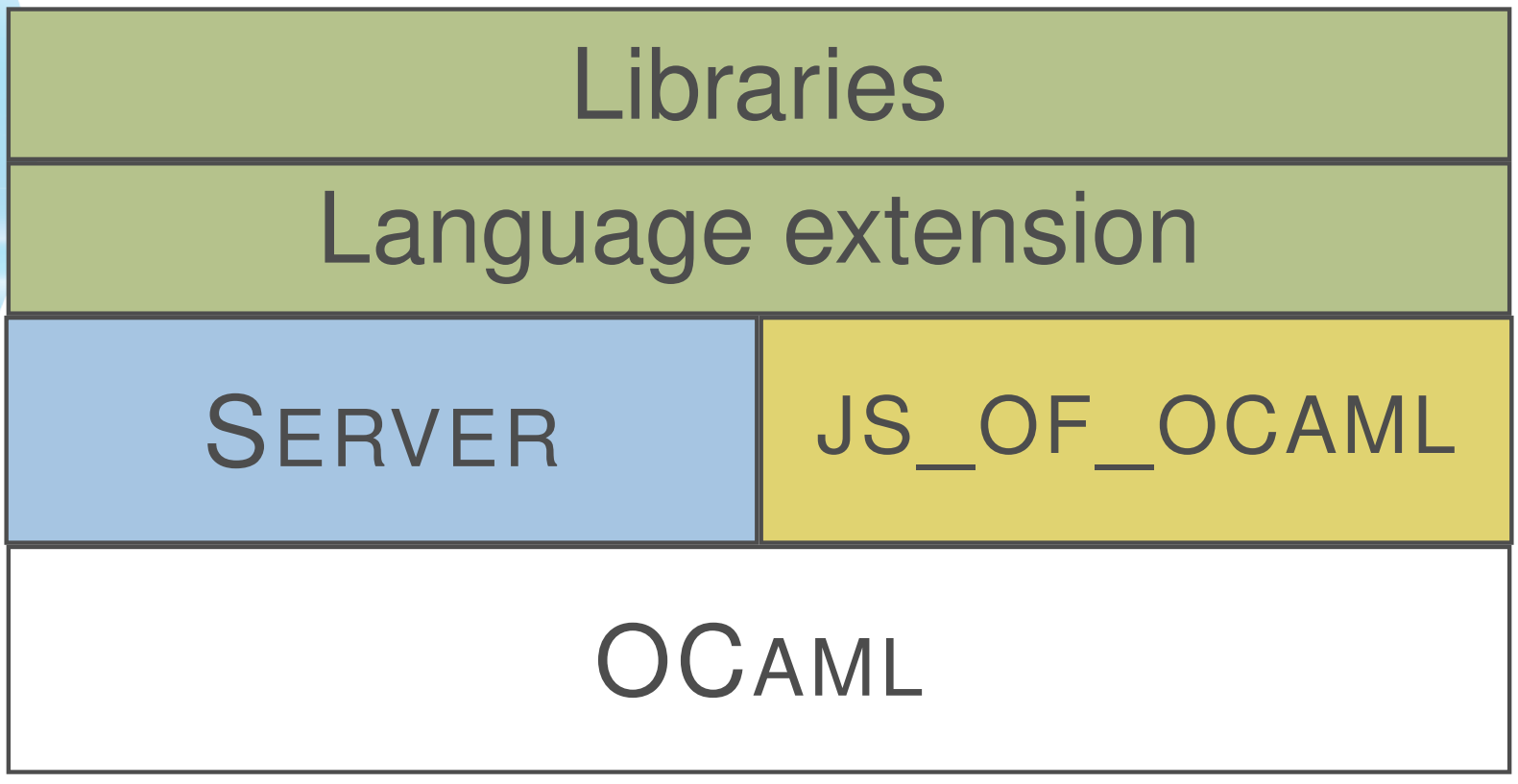
The OCSIGEN project



The OCSIGEN project

ocsigen

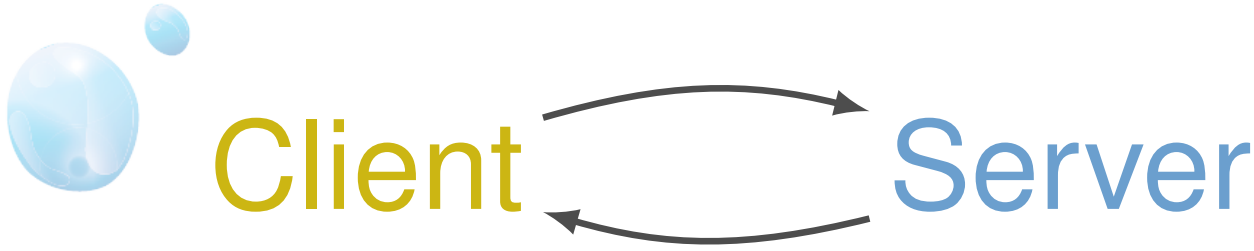
fresh air in web programming



1. ELIOM's language extension

2. Formalization
   - Execution
   - Compilation

Client and Server annotations



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

```
1 let%server s = ...
2
3 let%client c = ...
4
5 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.

```
1 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.

```
1 let%server s : int = 2
2
3 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
```

A button example

button.eliom

```
1 let%server hint_button msg =
2   button
3     ~a:[a_onclick [%client fun _ -> alert ~%msg] ]
4     [text "Show hint"]
```

A button example

button.eliom

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

button.html

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

A button example

button.eliom

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

button.html

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

button.eliomi

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

ELIOM$_\varepsilon$

Grammar:

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

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

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

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

$$c_s \in \mathit{Const}_s \qquad c_c \in \mathit{Const}_c \qquad \text{(Constants)}$$

Types:

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

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

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

Meta-syntactic variables:

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

Example

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

$$\texttt{let}_s\ s : int_s = 2\ \texttt{in}$$

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

$$\ldots$$

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!

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

Given the predefined converters:

$$\texttt{cint} : \mathit{int}_s \rightsquigarrow \mathit{int}_c$$
$$\texttt{fragment} : \forall\alpha.(\{\alpha\} \rightsquigarrow \alpha)$$

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!

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

Given the predefined converters:

$$\texttt{cint} : \mathit{int}_s \rightsquigarrow \mathit{int}_c$$
$$\texttt{fragment} : \forall\alpha.(\{\alpha\} \rightsquigarrow \alpha)$$

Example with converters

```
1 let%server x : int fragment = [%client 1 + 3 ]
2
3 let%client c : int = 3 + ~%x
```

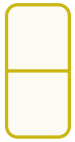
$$\texttt{let}_s\ x : \{\text{int}_c\} = \{\{\ 1+3\ \}\}\ \texttt{in}$$
$$\texttt{let}_c\ y : \text{int}_c = 3 + \texttt{fragment\%}x\ \texttt{in}$$
$$(y : \text{int}_c)$$

Example of execution

ELIOM code

```
let_s x = {{ 1 + 3 }} in
let_c y = 3 + fragment%x in
y
```

Queue



Example of execution

ELIOM code

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

ELIOM code

y

Queue

$y = 7$

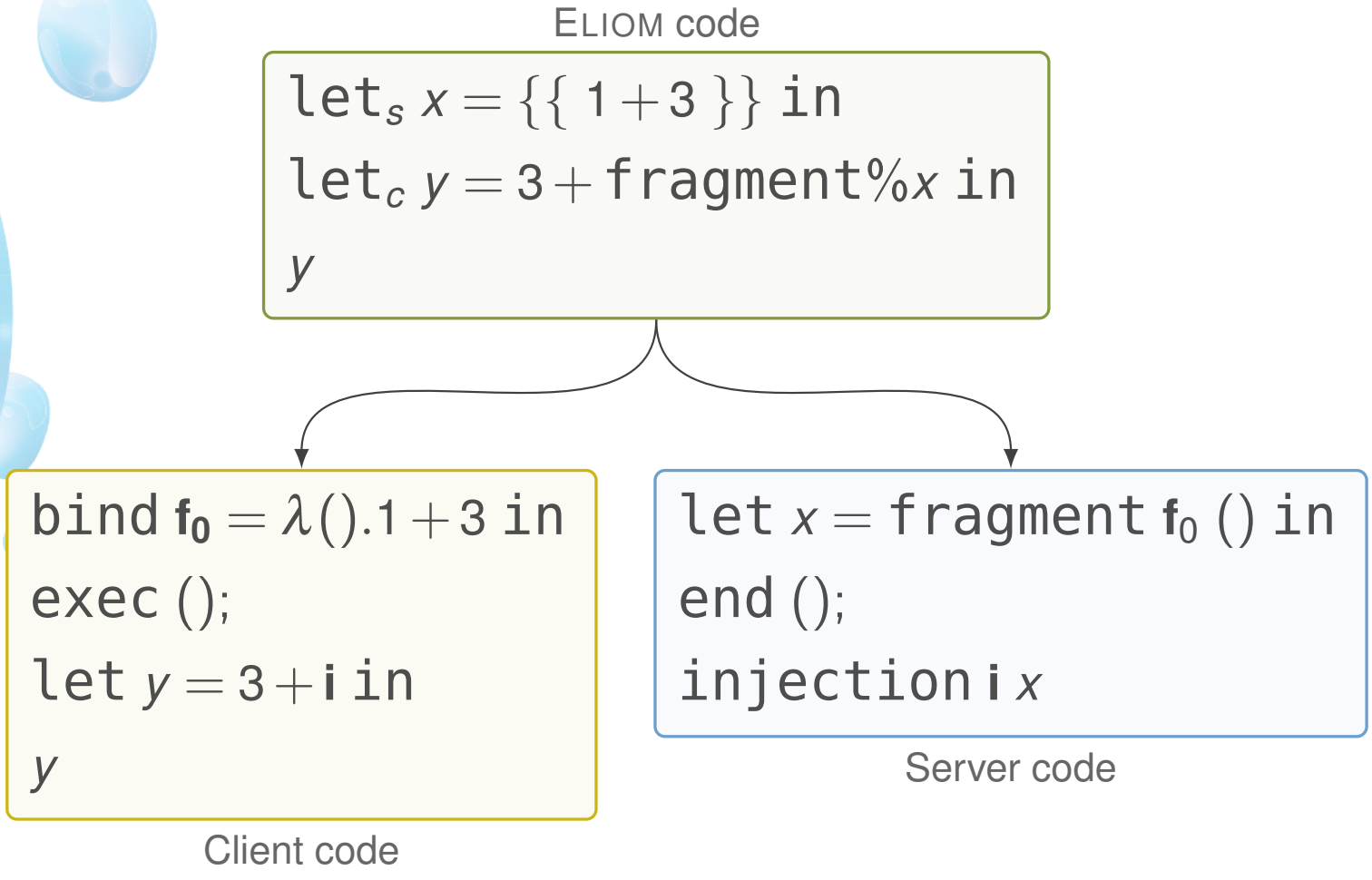
Example of execution

ELIOM code

7

Queue

Example of compilation



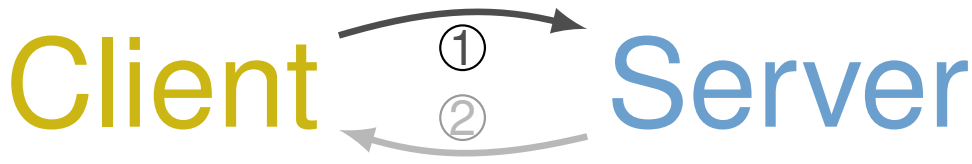
Example of compilation

```
bind f0 = λ().1 + 3 in
exec ();
let y = 3 + i in
y
```

Client code

```
let x = fragment f0 () in
end ();
injection i x
```

Server code



Queue

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

Injections

$i \mapsto \mathbf{r}$

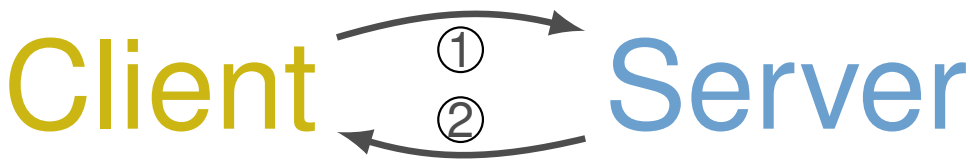
Example of compilation

```
bind f0 = λ().1 + 3 in
exec ();
let y = 3 + i in
y
```

Client code

```
let x = fragment f0 () in
end ();
injection i x
```

Server code

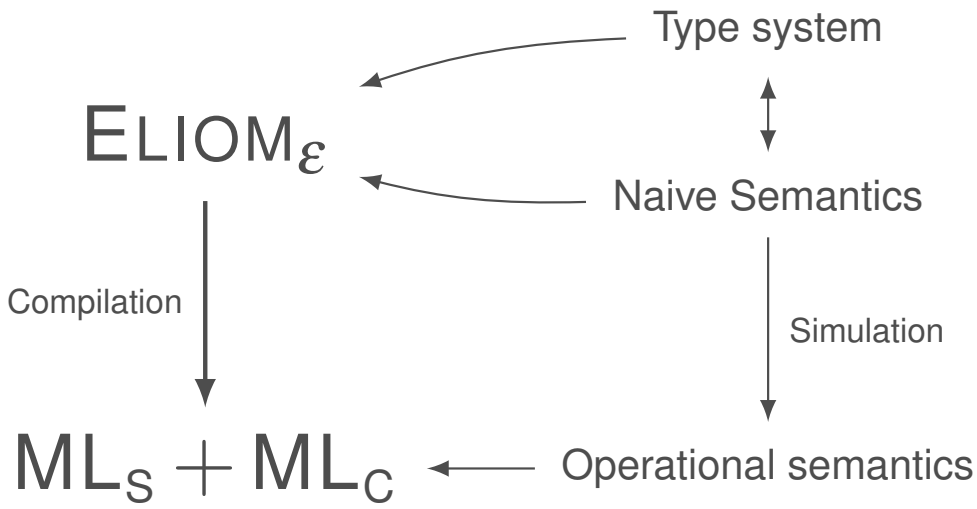


Queue

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

Injections

$i \mapsto \mathbf{r}$


Type system
ELIOM$_\varepsilon$
Naive Semantics
Compilation
Simulation
ML$_S$ + ML$_C$
Operational semantics

Summary

We developed an extension of ML with:

- A type system that allows tracking of locations
- Typesafe client-server communication via converters
- An efficient evaluation strategy that avoids too many communications
- A compilation scheme preserving that evaluation strategy

In the paper:

- Details of the type system and semantics
- Theorems for Soundness and Simulation

All of this is implemented and used: `https://ocsigen.org`

Ongoing and Future work

- Extension to the module system
- Server datatypes parametrized by client types
- A modified OCaml compiler for Eliom:
  WIP version at
  `https://github.com/ocsigen/ocaml-eliom`

Questions ?

Type system

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

VAR

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

FRAGMENT

$$\frac{\Gamma \triangleright_c e_c : \tau_c}{\Gamma \triangleright_s \{\{\ e_c\ \}\} : \{\tau_c\}}$$

INJECTION

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

One predefined constant types: `serial`

Two predefined converters:

$$\texttt{serial} : \texttt{serial} \rightsquigarrow \texttt{serial}$$

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

Type system

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

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

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

One predefined constant types: `serial`

Two predefined converters:

$$\texttt{serial} : \texttt{serial} \rightsquigarrow \texttt{serial}$$

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

Type system

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

VAR

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

FRAGMENT

$$\frac{\Gamma \triangleright_c e_c : \tau_c}{\Gamma \triangleright_s \{\{\ e_c\ \}\} : \{\tau_c\}}$$

INJECTION

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

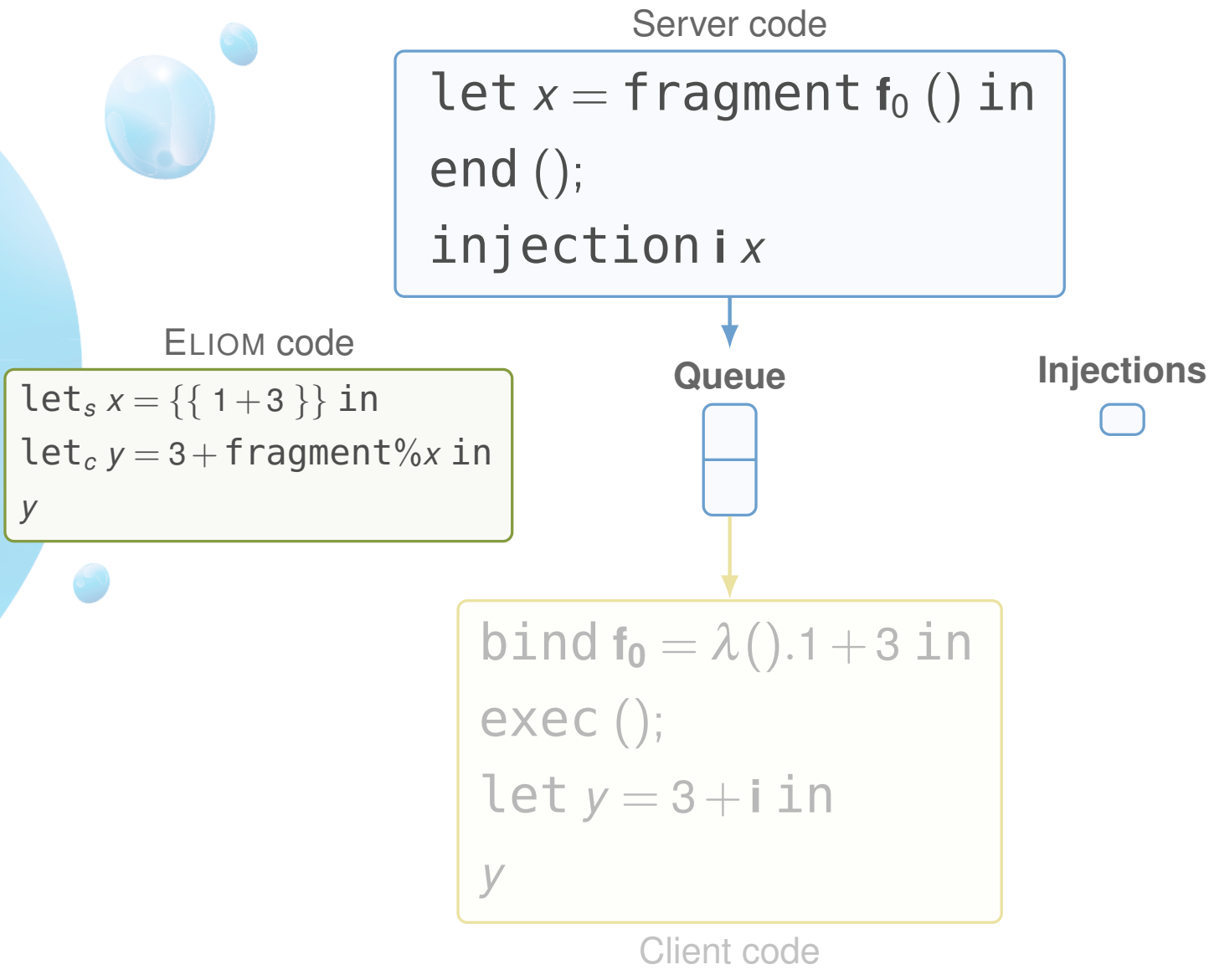
One predefined constant types: `serial`

Two predefined converters:

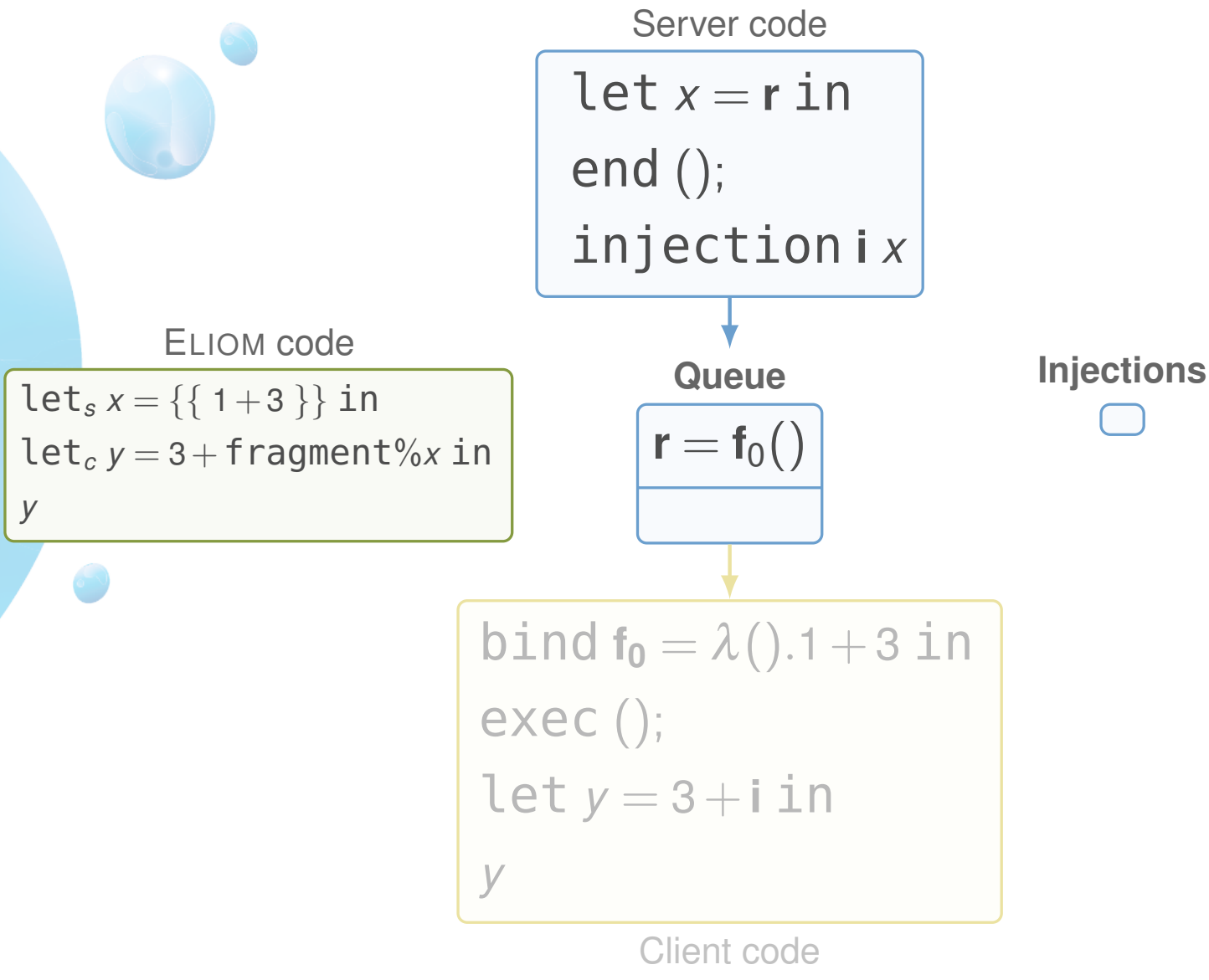
$$\texttt{serial} : \texttt{serial} \rightsquigarrow \texttt{serial}$$

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

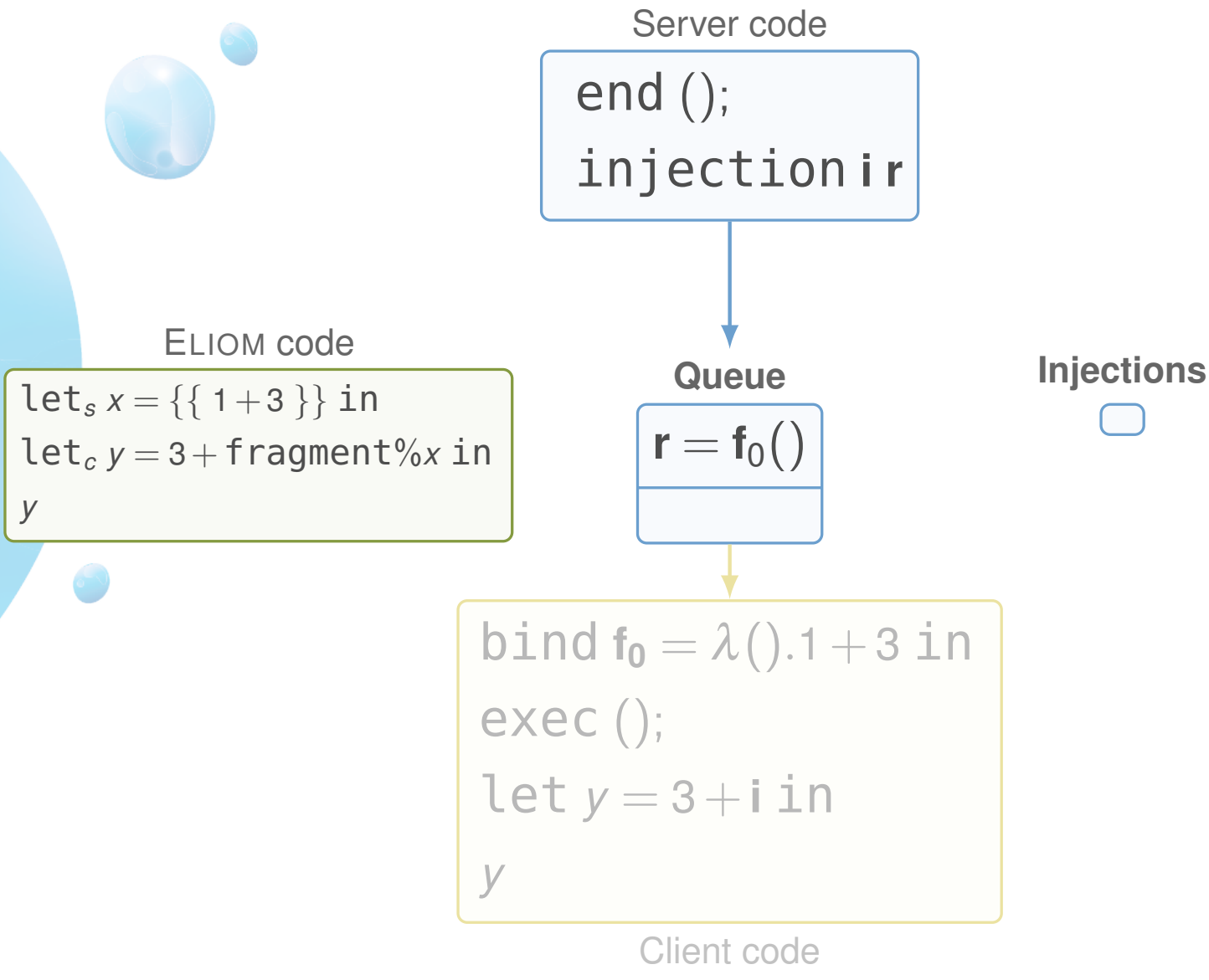
Execution of the compiled code



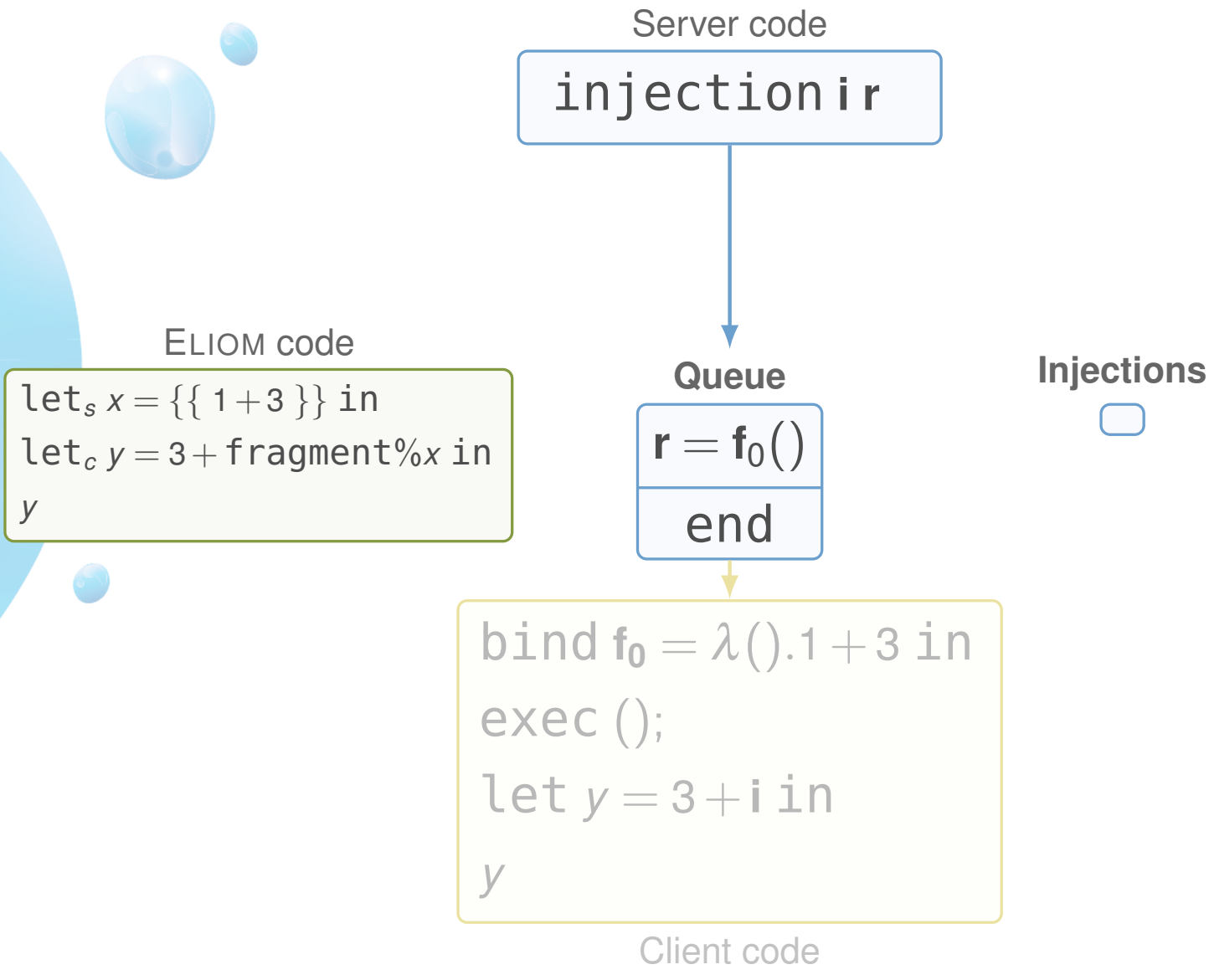
Execution of the compiled code



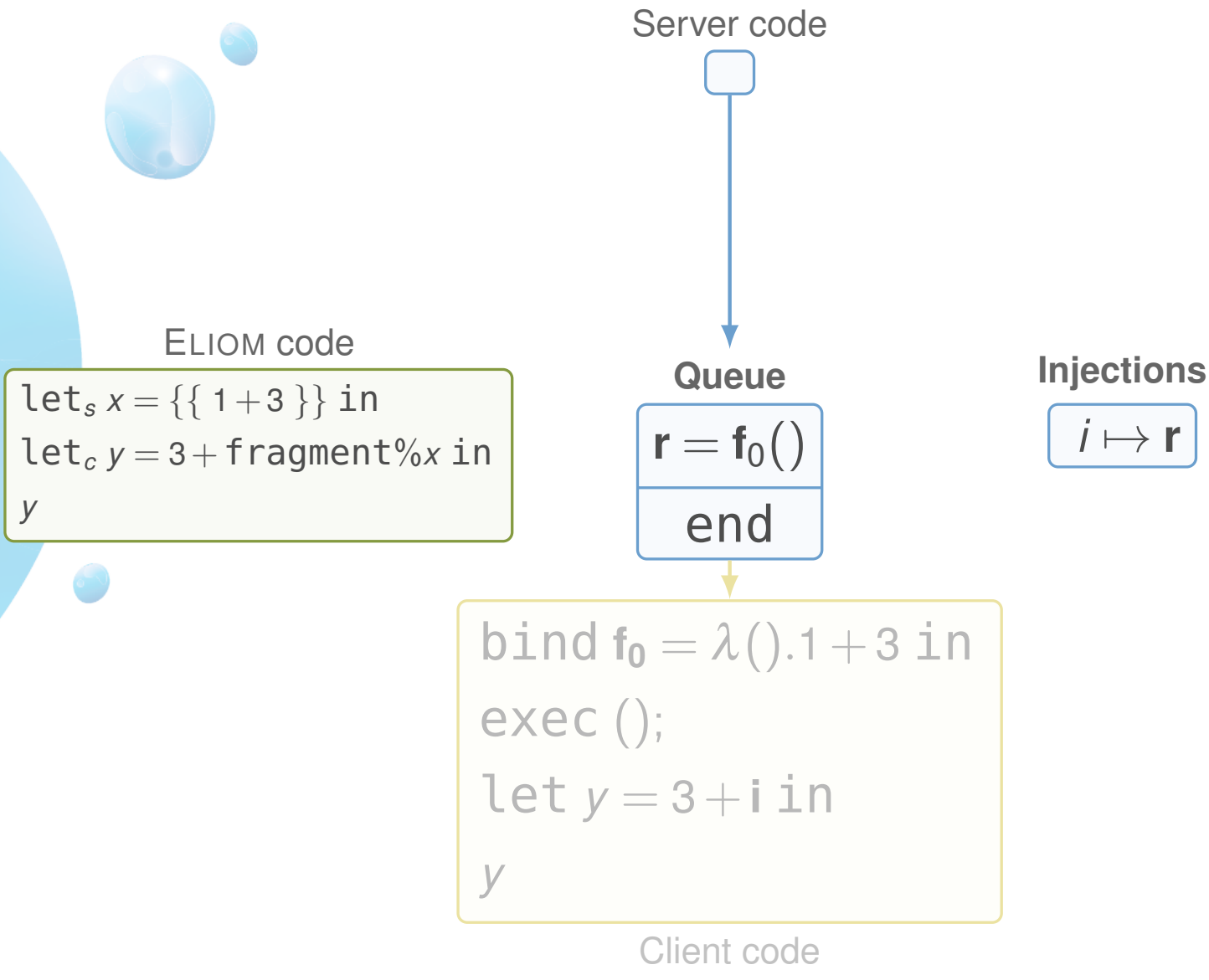
Execution of the compiled code



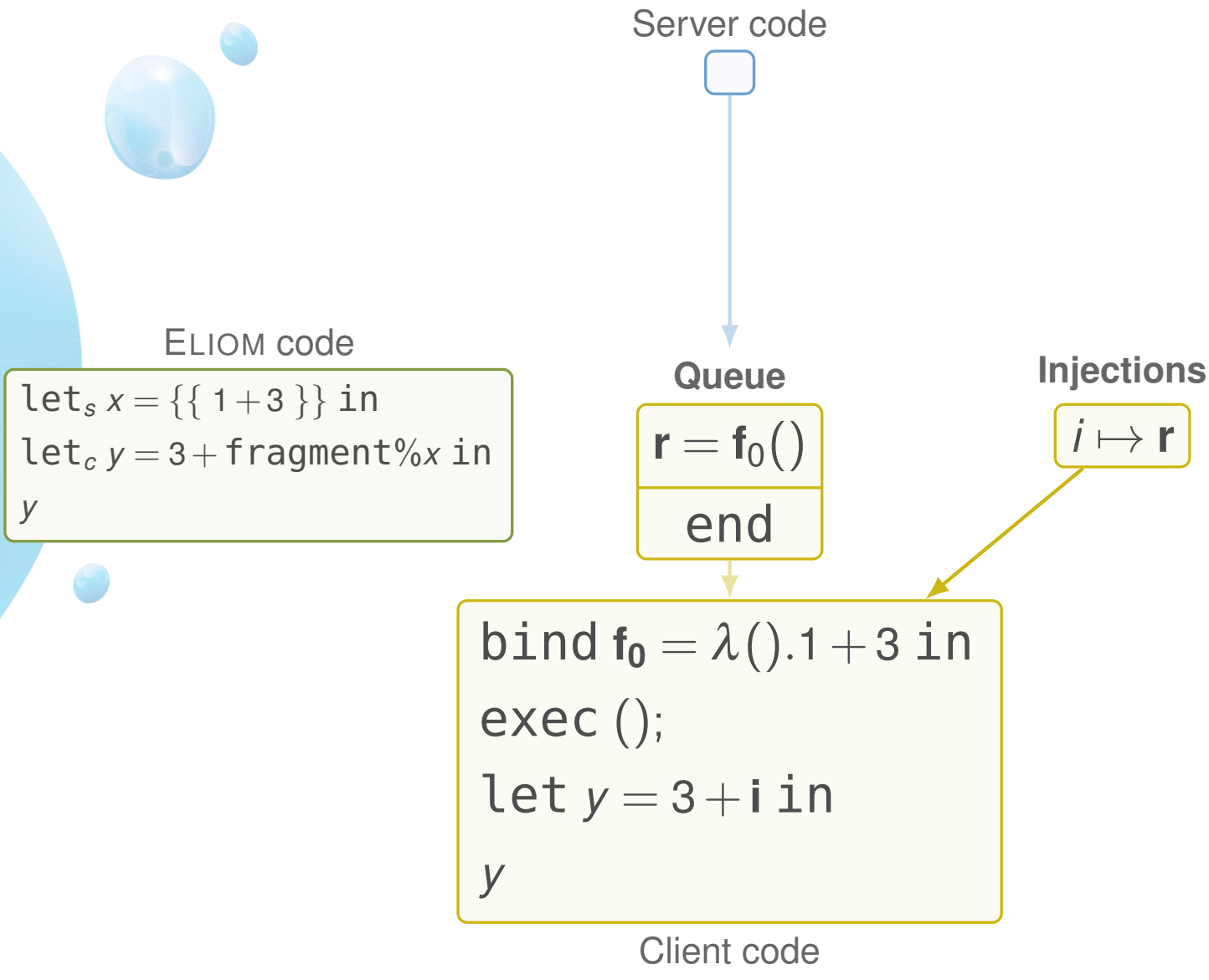
Execution of the compiled code



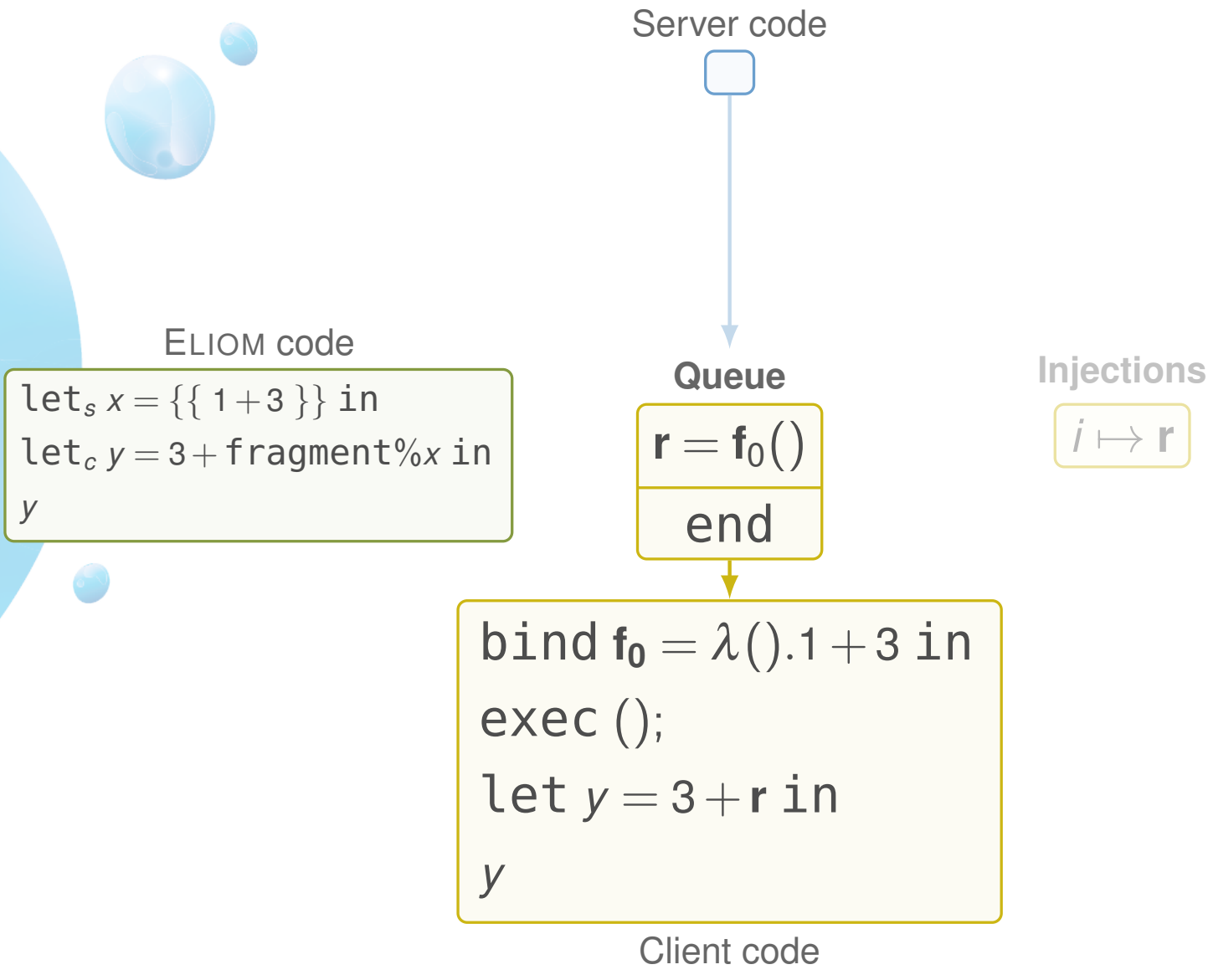
Execution of the compiled code



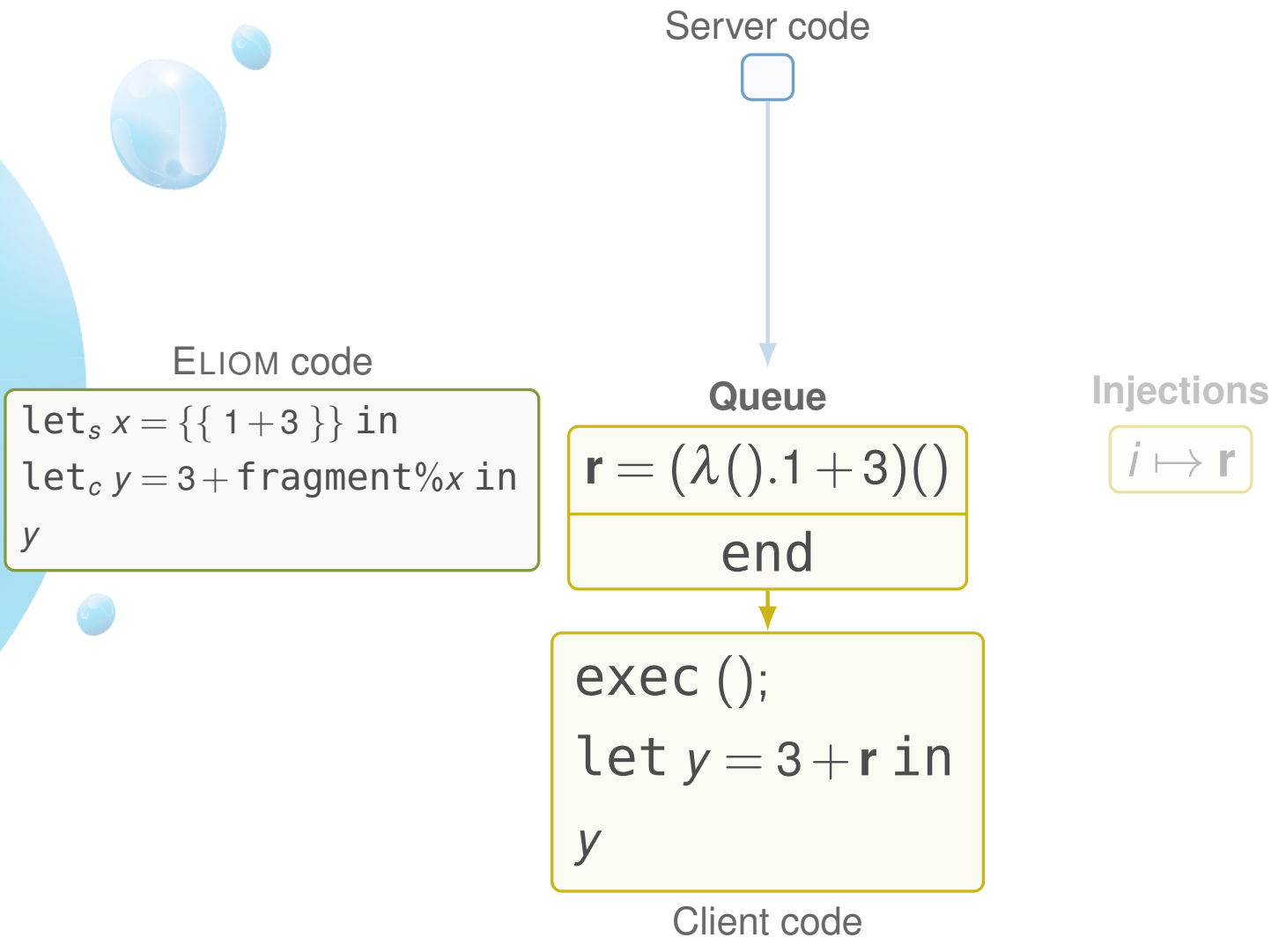
Execution of the compiled code



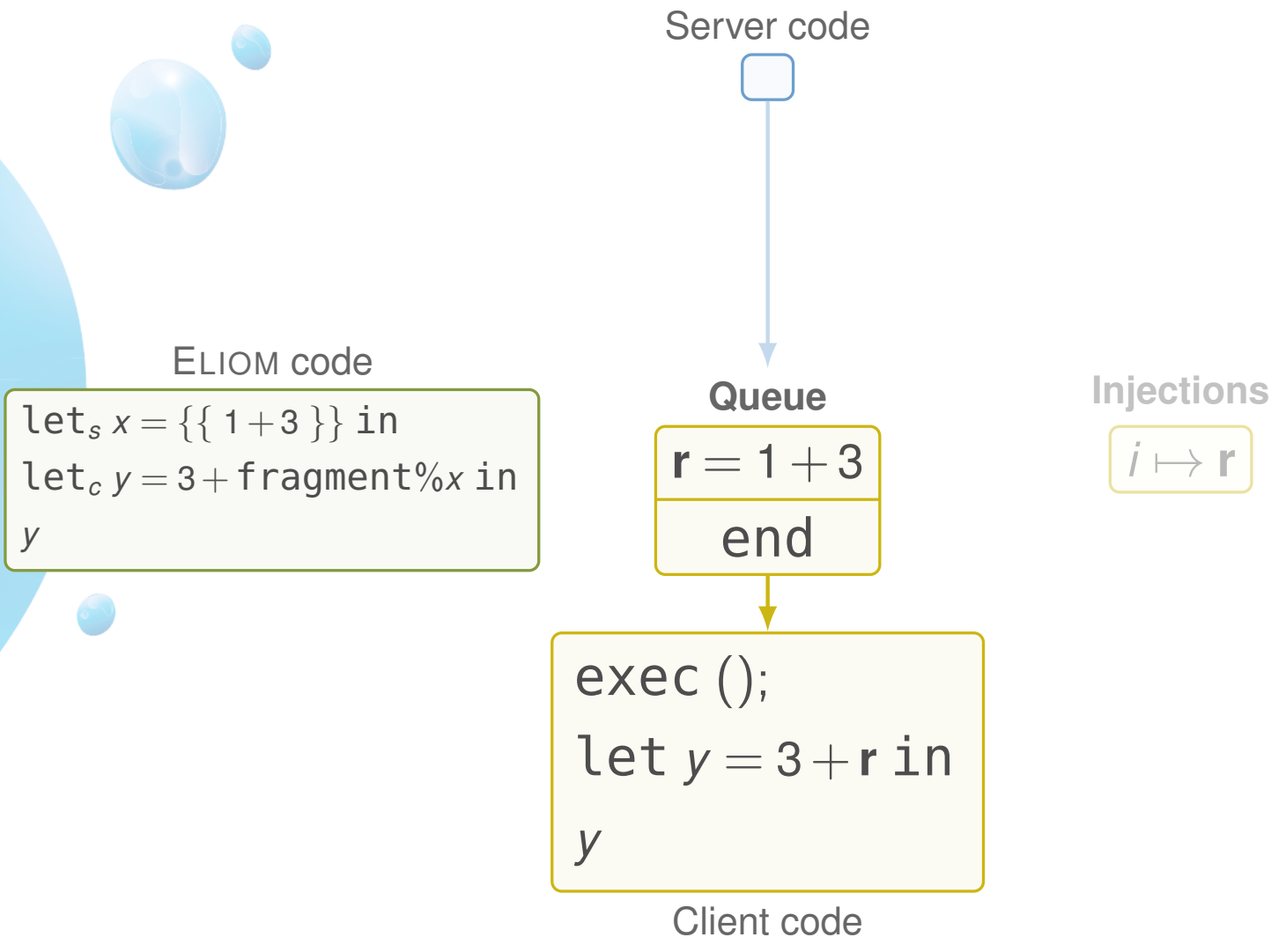
Execution of the compiled code



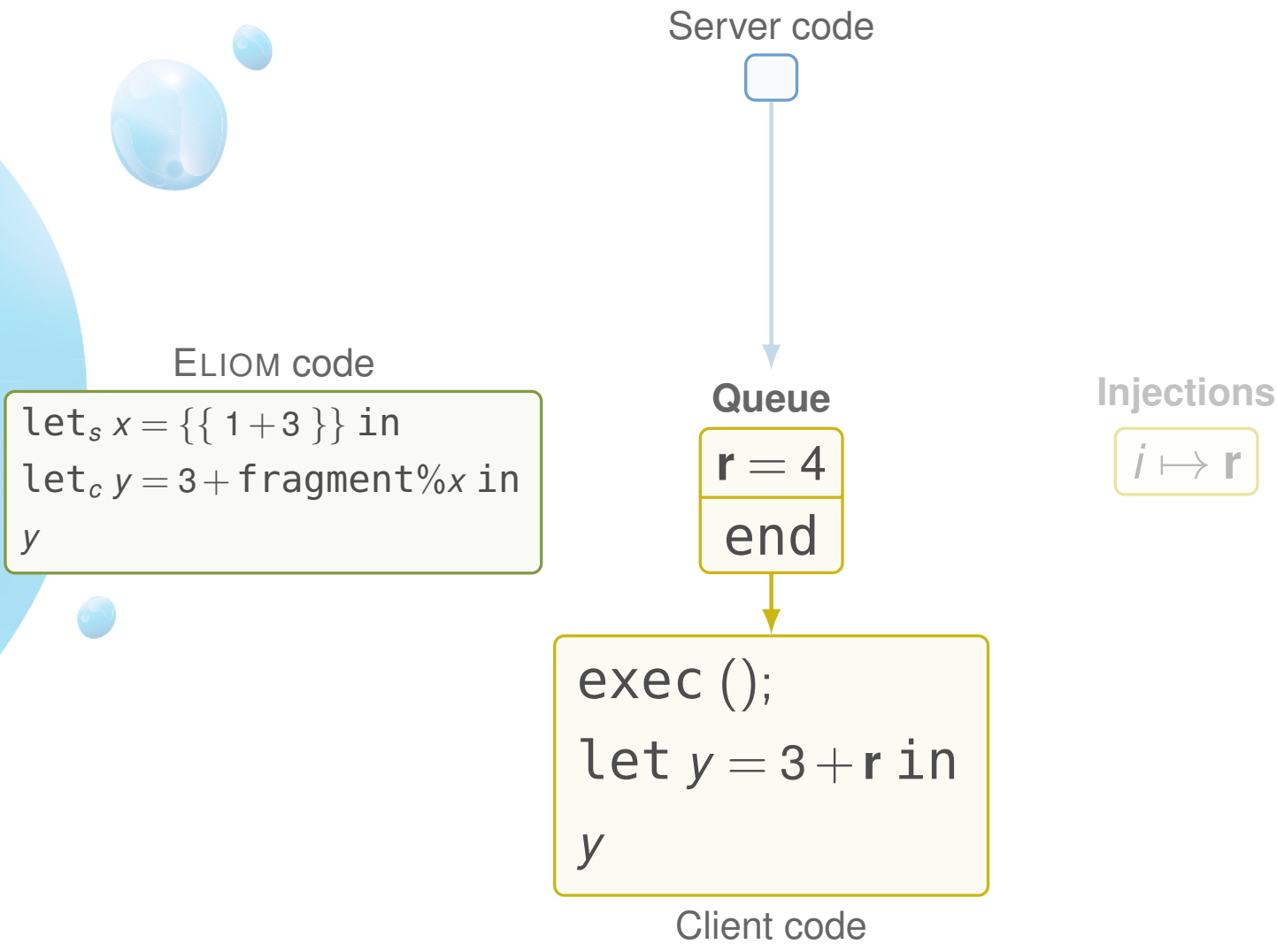
Execution of the compiled code



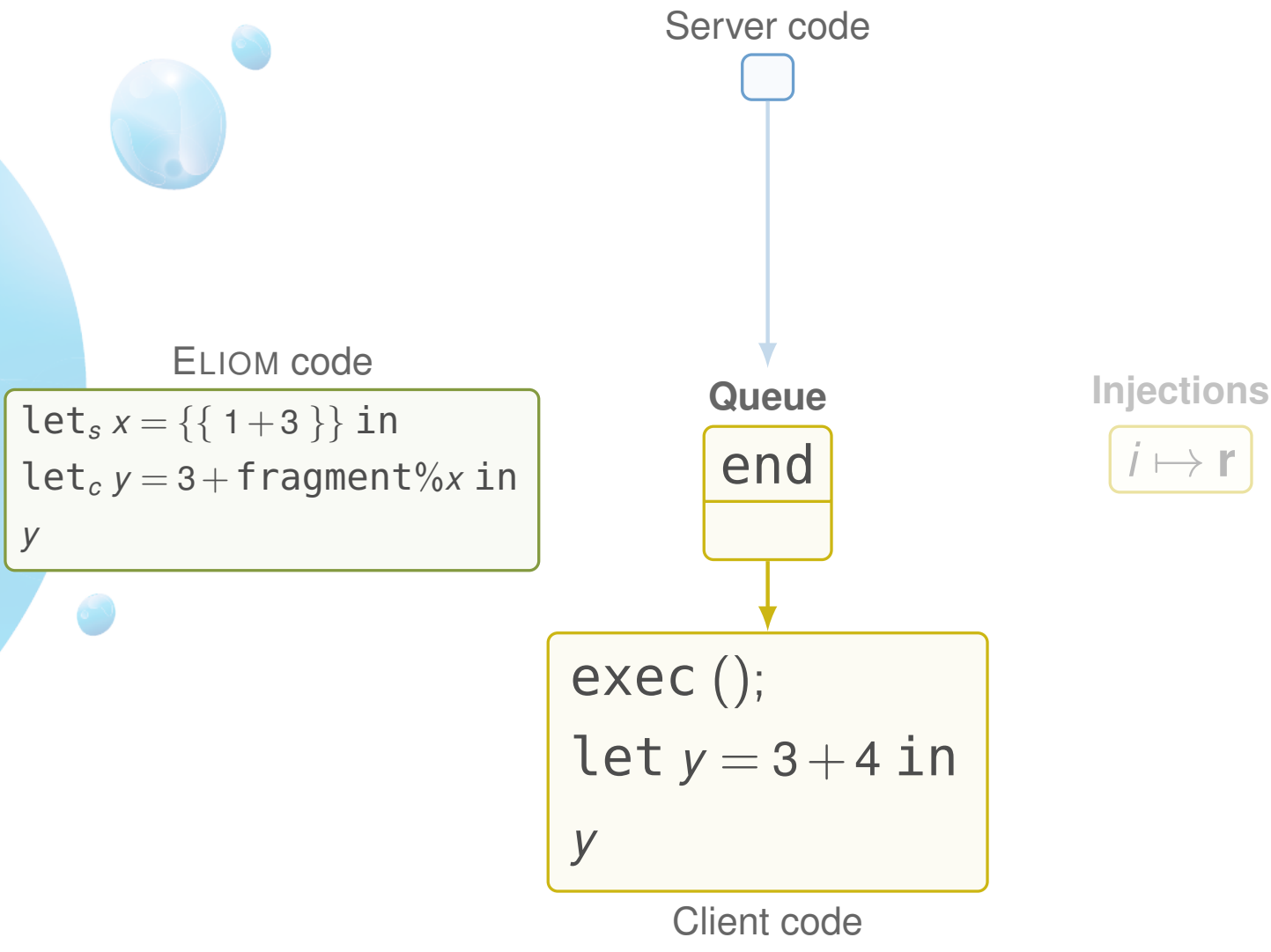
Execution of the compiled code



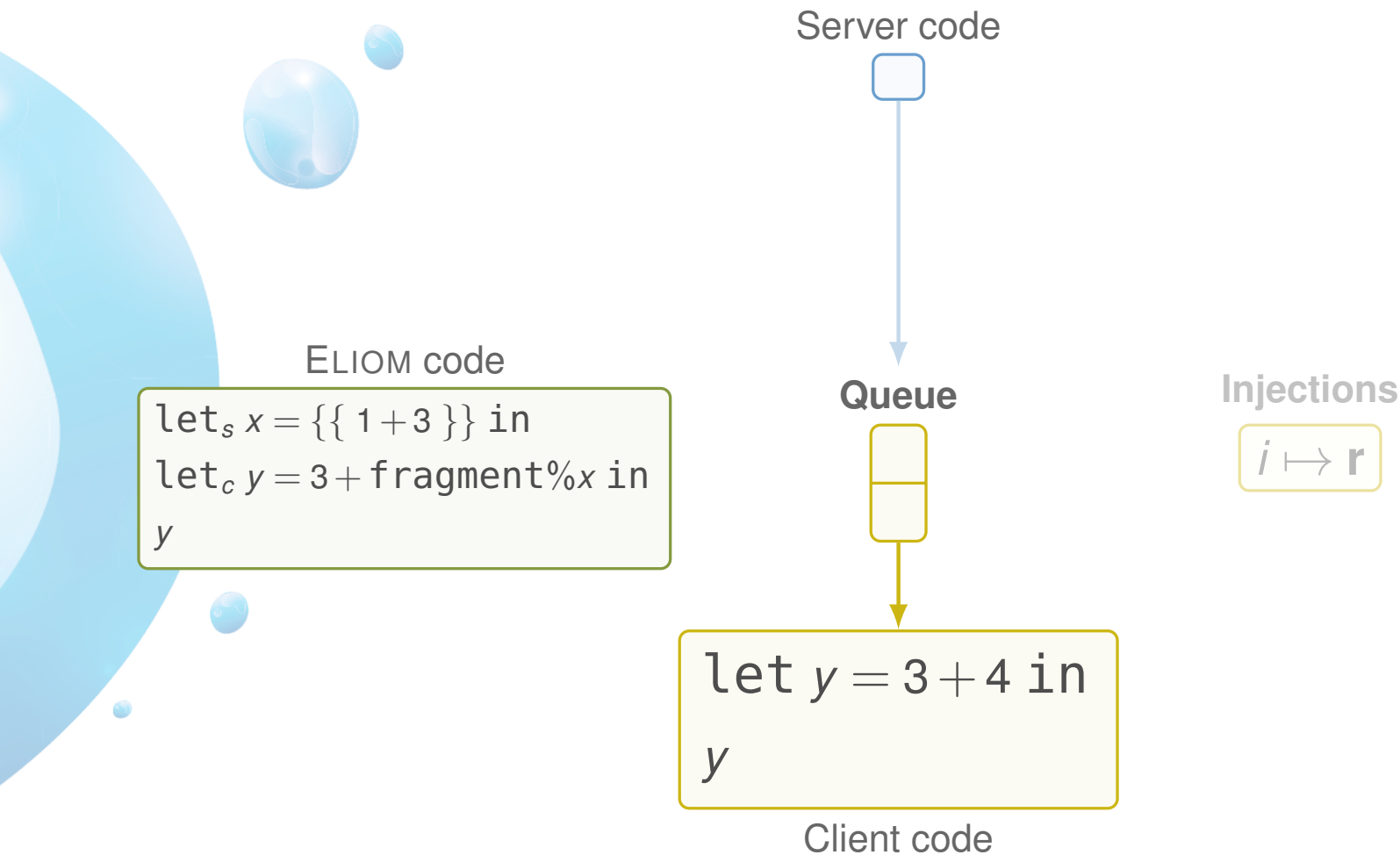
Execution of the compiled code



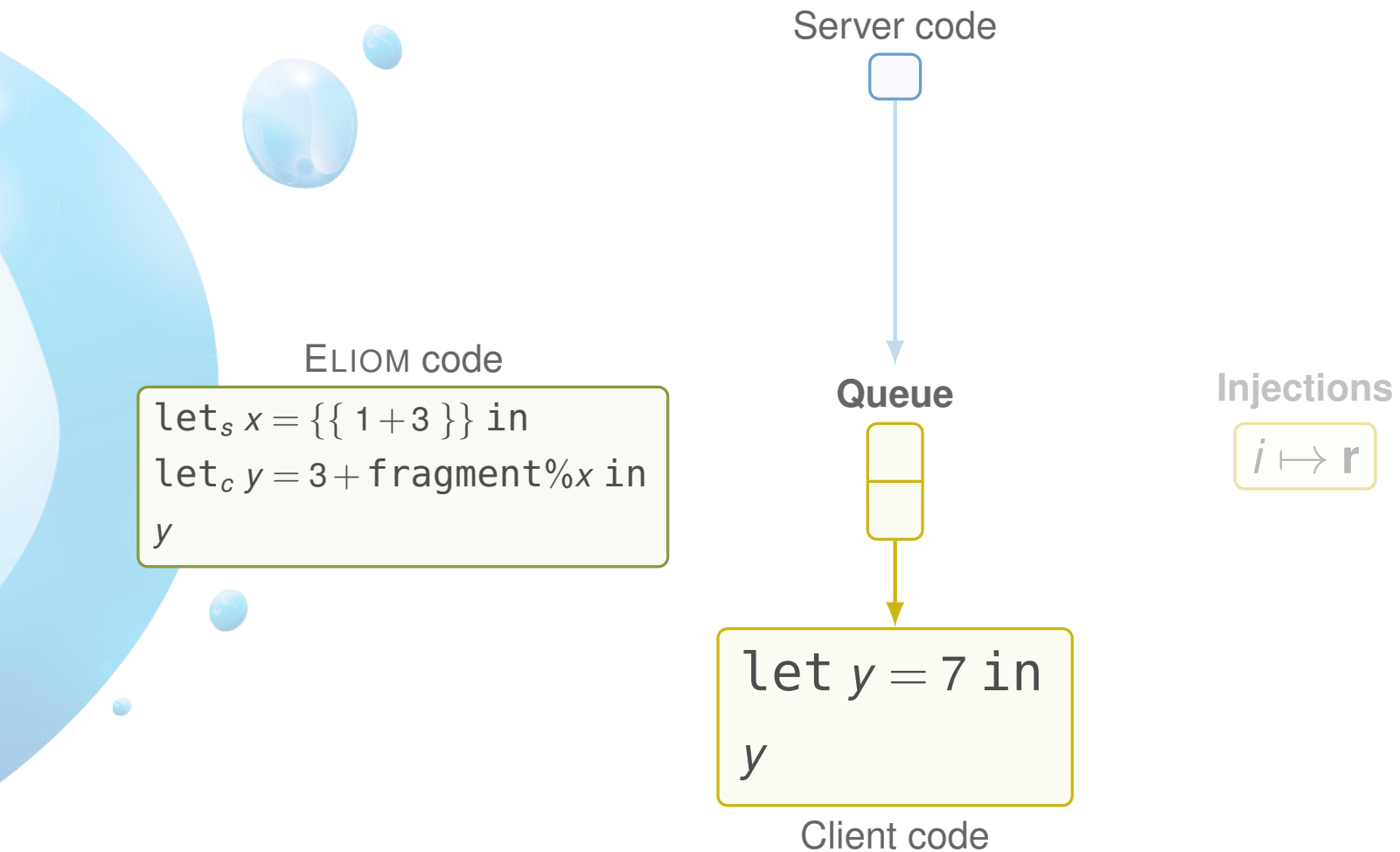
Execution of the compiled code



Execution of the compiled code



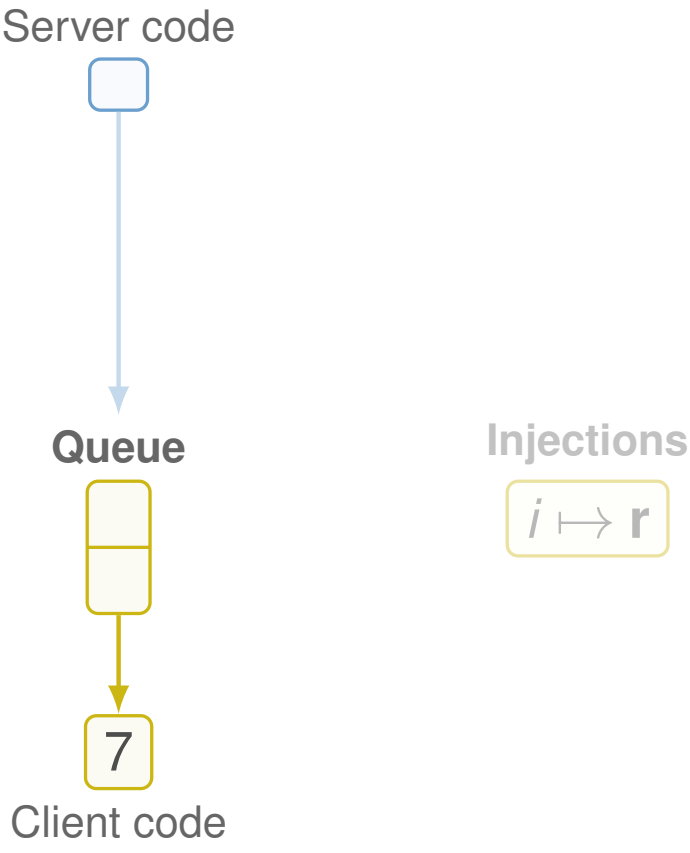
Execution of the compiled code



Execution of the compiled code

ELIOM code

```
let_s x = {{ 1+3 }} in
let_c y = 3+fragment%x in
y
```



Injections

$i \mapsto \mathbf{r}$