Eliom: tierless Web programming from the ground up

Gabriel Radanne
IRIF UMR 8243 CNRS
Univ Paris Diderot, Sorbonne Paris Cité
gabriel.radanne@irif.fr

Vasilis Papavasileiou
IRIF UMR 8243 CNRS
Univ Paris Diderot, Sorbonne Paris Cité
vasilis@fastmail.net

Jérôme Vouillon
IRIF UMR 8243 CNRS
Univ Paris Diderot, Sorbonne Paris Cité
BeSport
CNRS
jerome.vouillon@irif.fr

Vincent Balat
IRIF UMR 8243 CNRS
Univ Paris Diderot, Sorbonne Paris Cité
BeSport
vincent.balat@irif.fr

ABSTRACT

Eliom is a dialect of OCaml for Web programming. It can be used both server and client-side. Server and client sections can also be mixed in the same file using syntactic annotations. This allows one to build a whole application as a single program, in which it is possible to define in a composable way reusable widgets with both server and client behaviors. Our language also enables simple type-safe communication. Eliom matches the specificities of the Web by allowing the programmer to interleave client and server code while maintaining efficient one-way server-to-client communication.

We present how the language extensions introduced by Eliom enable a new paradigm for Web programming, and how this paradigm allows building complex libraries easily, safely, and in a composable manner.

CCS CONCEPTS

•**Software and its engineering → Functional languages;** *Concurrent programming languages;*

KEYWORDS

Web, client/server, OCaml, ML, Eliom, functional

ACM Reference format:
Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: tierless Web programming from the ground up. In *Proceedings of IFL 2016, Leuven, Belgium, August 31-September 2, 2016,* 12 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064901

1 INTRODUCTION

The emergence of rich Web applications has led to new challenges for programmers. Most early Web applications followed a simple model: use the language of your choice to create, on the server, a Web page composed of HTML for structure, CSS for styling, and JavaScript for interactivity, and send all this data to the client using HTTP. This model does not stand up to the requirements of the modern Web. For example, current applications involve complex behaviors that rely on bi-directional communication between clients and servers (*e.g.*, notifications and messaging). Such communication patterns are not easy to achieve while maintaining a strict separation between client- and server-side logic, let alone in a type-safe way. Additionally, the tendency towards larger Web applications imposes composability requirements that go beyond the capabilities of early Web technologies.

Recent work proposes languages for expressing the client-side and the server-side code in a unified way, such as Links (Cooper et al. 2006) and Ur/Web (Chlipala 2015a,b). These *tierless* languages can accomodate the communication patterns of the modern Web, and provide encapsulation and composition of components that involve both client and server behaviors. Tierless languages can be statically typed, providing guarantees for each side individually, but also for the communication between them.

Our paper discusses Eliom, which is an extension of OCaml that can express client- and server-side code side-by-side. Eliom provides the encapsulation and composability advantages of tierless programming, and additionally brings in the benefits of an existing language. Concretely, Eliom users have direct access to the mature ecosystem of OCaml libraries. Additionally, Eliom programs benefit from the very rich type system of OCaml, extended to reason about the client-server boundary. The Eliom-specific primitives (which we describe in Section 2) are limited in scope and orthogonal to the standard constructs of an ML-like language. This separation of concerns allows us to reason about Eliom formally (Radanne et al. 2016).

Eliom is part of the larger Ocsigen (Balat et al. 2009) project. Ocsigen provides a comprehensive set of tools and libraries for developing Web applications in OCaml, including the compiler js_of_ocaml (Vouillon and Balat 2014), a Web server, and libraries for concurrency (Vouillon 2008), HTML manipulation (TyXML 2017) and database interaction (Scherer and Vouillon 2010), Ocsigen libraries take deep advantage of the OCaml type system to provide guarantees about various aspects of client- and server-side Web programming, *e.g.*, this paper shows examples in which we produce HTML whose validity is guaranteed by the type system (TyXML

This work was partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org.

2017). These guarantees are complementary to the ones that Eliom provides on client-server communication.

Our language primitives coupled with preexisting OCaml libraries (such as the ones provided by Ocsigen) have allowed us to build a comprehensive tierless Web framework. This article elaborates on this framework with an emphasis on its links to the Eliom programming language. Section 3 relies on interesting parts of the Eliom library to demonstrate that our minimalist primitives suffice for implementing all the abstractions needed to support a tierless Web development style, for instance higher-level communication mechanisms. At the same time, the paper serves as a practical introduction to programming with Eliom, *e.g.*, by demonstrating how our programming paradigm allows expressing complex widgets with client-server behaviors in very few lines of code.

Our core design decision of building on an existing language additionally permits efficient implementation, as we discuss in Section 4. Specifically, we have implemented an Eliom compiler as an unobtrusive extension of the OCaml compiler. Our compiler produces server and client-code that retains the performance characteristics of OCaml. More generally, all OCaml development tools adapt to Eliom with small-scale modifications, or without any modifications at all.

2 A GLIMPSE OF THE ELIOM LANGUAGE

An Eliom application is composed of a single program which is decomposed by the compiler into two parts. The first part runs on a Web server and manages several sessions at the same time, possibly sharing data between sessions and keeping state for each browser or tab currently running the application. The client program, compiled statically to JavaScript, is sent to each client by the server program along with the HTML page, in response to the initial HTTP request.

Composition. The Eliom language allows to define and manipulate *on the server*, as first class values, fragments of code which will be executed *on the client*. This gives us the ability to build reusable widgets that capture both the server and the client behaviors transparently. This makes it possible to define client-server building blocks (and libraries thereof) without explicit support from the language. For instance, in the case of Eliom, RPCs, a functional reactive library for Web programming, and a GUI toolkit (Ocsigen Toolkit 2017) have all been implemented as libraries.

Explicit communication. Eliom is using manual annotations to determine whether a piece of code is to be executed server- or client-side (Balat 2013; Balat et al. 2012). This design decision stems from our belief that the programmer must be well aware of where the code is to be executed, to avoid unnecessary remote interaction. Explicit annotations also prevent ambiguities in the semantics, allow for more flexibility, and enable the programmer to reason about where the program is executed and the resulting trade-offs. Programmers can thus ensure that some data stays on the client or on the server, and choose how much communication takes place.

A simple and efficient execution model. Eliom relies on a novel and efficient execution model for client-server communication that avoids back-and-forth communication. This model is simple and predictable. Having a predictable execution model is essential in the context of an impure language, such as OCaml.

We now present the language extension that deals with client-server code and the corresponding communication model. Even though Eliom is based on OCaml, little knowledge of OCaml is required. We explicitly provide some type annotations for illustration purposes, but they are not mandatory.

2.1 Sections

The location of code execution is specified by *section* annotations. We can specify whether a declaration is to be performed on the server or on the client as follows:

```
let%server s = ...
let%client c = ...
```

A third kind of section, written as **shared**, is used for code executed on both sides.

We use the following color convention: client is in **yellow**, server is in **blue** and shared is in **green**. Colors are however not mandatory to understand the rest of this paper.

2.2 Client fragments

A client-side expression can be included inside a server section: an expression placed inside `[%client ... ]`will be computed on the client when it receives the page; but the eventual client-side value of the expression can be passed around immediately as a black box on the server. These expressions are called client *fragments*.

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

For example, here, the expression `1 + 3` will be evaluated on the client, but it's possible to refer server-side to the future value of this expression (for example, put it in a list). The value of a client fragment cannot be accessed on the server.

The type `'a fragment` presented here is an applicative functor (see the definition of `fmap` bellow) but not a monad. We discuss this fact in the state-of-the art section, as it is a distinctive feature compared to various other frameworks.

```
let fmap
  : ('a -> 'b) fragment -> 'a fragment -> 'b fragment
  = fun f x -> [%client ~%f ~%x ]
```

2.3 Injections

Values that have been computed on the server can be used on the client by prefixing them with the symbol `~%`. We call this an *injection*.

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

Here, the expression `1 + 2` is evaluated and bound to variable `s` on the server. The resulting value 3 is transferred to the client together with the Web page. The expression `~%s + 1` is computed client-side.

An injection makes it possible to access client-side a client fragment which has been defined on the server:

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

The value inside the client fragment is extracted by `~%x`, whose value is 4 here.

3 USING ELIOM

We now provide examples that demonstrate how the language features of Section 2 can be used to build HTML pages with dynamic behavior in a composable fashion. We show how to create pieces of HTML pages, but also libraries that are useful for Web programming.

Our examples are extracted from code that appears in the Ocsigen tutorial (Tutorial 2017) and in the Eliom library (Eliom 2017). Each example was chosen to illustrate a particular new programming construct that is used pervasively in the Eliom ecosystem.

3.1 Client-server behaviors

Our first example demonstrates how Eliom allows to mix client and server behaviors in a given function. We create a button that increments a client-side counter and invokes a callback each time it is clicked. This widget is produced by the function `counter` below. This function uses an HTML DSL (TyXML 2017) that provides combinators such as `button` and `a_onclick` (which respectively create an HTML tag and an HTML attribute). See Section 3.4.1 for more details on this DSL. The `~a` is the OCaml syntax for named arguments. Here, it is used for the list of HTML attributes.

The example uses a handler for the `onclick` event: since clicks are performed client-side, this handler needs to be a client function. This client function modifies the widget's state (the client-side reference `state`) and then calls the user-provided client-side callback `action`. This demonstrates that the higher-order nature of OCaml can be used in our client-server setting, and that it is useful for building server-side Web page fragments with parameterized client-side behaviors. In addition, note that the separation between `state` and `action` makes it straightforward to extend this example with a second button that decrements the counter while sharing the associated state.

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

The server widget `counter` captures both server and client behavior. The behavior is properly encapsulated inside the widget. Here is the corresponding API for such a widget:

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

This widget is easily composable: the embedded client state cannot affect nor be affected by any other widget; and it can be used to build larger widgets.

3.1.1 Client-server communication. Our counter widget showcases complex patterns of interleaved client and server code, including passing client fragments as arguments to server functions, and subsequently to client code. This would be costly if the communication between the client and the server were done naively.

Eliom employs an efficient communication mechanism. Specifically, the server only ever sends data along with the initial version of the page. This is made possible by the fact that client fragments are not executed immediately when encountered inside server code. Intuitively, the semantics—presented formally in (Radanne et al. 2016)—is the following: when the server code is executed, the encountered client code is not executed right away; instead, it is just registered for later execution, once the Web page has been sent to the client. Only then is all the client code executed in the order it was encountered on the server.

In addition to being efficient, our predictable execution order allows the programmer to reason about Eliom programs, especially in the presence of side effects, without being intimately familiar with the details of the compilation scheme.

3.2 Heterogeneous datatypes

Some datatypes are represented in fundamentally different ways on the server and on the client. This is a consequence of the different nature of the server and the client environments. Eliom properly models this heterogeneous aspect by allowing to relate a client and a server datatype that share a similar semantics while having different definitions.

We use this feature to present a safe and easy to use API for remote procedure calls (RPCs).

3.2.1 Remote procedure calls. When using fragments and injections, the only communication taking place between the client and the server is the original HTTP request and response. However, further communication is sometimes desirable. A remote procedure call is the action of calling, from the client, a function defined on the server.

We present here an RPC API implemented using the Eliom language. The API is shown in Figure 1. An example can be seen in Figure 2.

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

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

Figure 1: The simplified RPC API

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

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

Figure 2: An example using the RPC API

In the example, we first create server-side an RPC endpoint using the function `Rpc.create`. Our example RPC adds 1 to its argument. The endpoint is therefore a value of type `(int,int)Rpc.t`, *i.e.*, an RPC whose argument and return values are both of type `int`. The type `Rpc.t` is abstract on the server, but is a synonym for a function type on the client. Of course, this function does not contain the actual implementation of the RPC handler, which only exists server-side.

To use this API, we leverage injections. By using an injection in `~%plus1`, we obtain *on the client* a value of type `Rpc.t`. We describe the underlying machinery that we leverage for converting RPC endpoints into client-side functions in Section 3.2.2. What matters here is that we end up with a function that we can call like any other; calling it executes the remote procedure call.

We now use the RPC API with the `counter` widget defined in Section 3.1. We assume the existence of a `save_counter` function, which saves the counter in a database, and of the `counter` function defined previously. We then proceed to define `save_counter_rpc` (*i.e.,* the server-side RPC interface for `save_counter`), and inject it into a fragment `f`. This fragment is subsequently used as the user-provided callback for `counter`. This way, each time the counter is incremented, its new value is saved server-side.

```
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
```

3.2.2 Converters. This ability to transform data before it is sent to the client via an injection is made possible by the use of *converters* (Radanne et al. 2016). Figure 3 broadly presents the converter API. Given a serialization format `serial`, a converter is a pair of a *server* serialization function and a *client* de-serialization function. Note that the client and server types are not necessary the same. Furthermore, we can arbitrarily manipulate the value before returning it. Several predefined converters are available for fragments, basic OCaml datatypes, and tuples in the module `Conv`. Implementation details about converters can be found in Section 4.5.

We can use converters to implement the RPC API (Figure 4). The server implementation of `Rpc.t` is composed of a handler, which is a server function, and a URL to which the endpoint answers. Our serialization function only sends the URL of the endpoint. The client de-serialization function uses this URL to create a function performing an HTTP request to the endpoint. This way, an RPC endpoint can be accessed simply with an injection. Thus, for the `create` function, we assume that we have a function `serve` of type `string -> (request -> answer)-> unit` that creates an HTTP handler at a specified URL. When `Rpc.create` is called, a unique identifier `id` is created, along with a new HTTP endpoint `"/rpc/id"` that invokes the specified function.

This implementation has the advantage that code using the `Rpc` module is completely independent of the actual URL used. The URL is abstracted away. Converters preserve abstraction by only exposing the needed information.

```
type%shared serial (* A serialization format *)
type%server ('a, 'b) converter = {
  serialize : 'a -> serial ;
  deserialize : (serial -> 'b) fragment }
```

Figure 3: Schematized API for 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 }
```

Figure 4: The simplified RPC implementation

```
val%server t
val%server create : url -> t
val%server send : t -> serial -> unit

val%client subscribe : url -> (serial -> unit) -> unit
```

Figure 5: `Broadcast`: Untyped broadcast API

```
type 'a event
(** Events with occurrences of type ['a] *)

val create : unit -> 'a event * ('a -> unit)
(** [create ()] returns an event [e] and a
    [send] function *)

val iter : ('a -> unit) -> 'a event -> unit
(** [iter f e] applies [f] to [e]'s occurrences *)
```

Figure 6: `Event`: Simplified reactive events API

```
type%server ('i, 'o) t
type%client ('i, 'o) t = 'o Event.event

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

Figure 7: `BroadcastEvent`: Shared reactive events API

3.2.3 Client-server reactive broadcasts. In the previous example, we used converters on rather simple datatypes: only a URL was sent, and a closure was created client-side. In this example, we use converters for a more ambitious API: lift *Functional Reactive Programming (FRP)* to be usable across client-server boundaries.

FRP is a paradigm that consists in operating on streams of data, either discrete (events) or continuous (signals). It has been used successfully to program graphical interfaces in a functional fashion, and can also be used to implement Web interfaces. Here, we show how to create an API that allows broadcasting server reactive events to a set of clients.

We assume pre-existent libraries implementing the following two APIs: An *untyped* broadcast API (Figure 5) and an FRP event API (Figure 6). Both of these APIs are orthogonal to Eliom's primitives; we can implement broadcast with standard Web techniques, and use the OCaml library (React 2017) for FRP events. The broadcast API operates on messages of type `serial`, the serialization type introduced in Figure 3.

Our goal is to produce a typed broadcast API shown in Figure 7. It is quite similar to the RPC API: we have a type `t` with different implementations on the client and the server, and a server function `create` that takes a converter and an event stream as argument

```
type%server ('i,'o) t = {
  conv : ('i, 'o) converter ;
  url : string ;
}

let%server create conv events =
  let url = "/broadcast/" ^ generate_new_id () in
  let t = Broadcast.create url in
  let send x =
    Broadcast.send t (conv.serialize x)
  in
  let () = Event.iter send serial_events in
  { conv ; url }

type%client ('i, 'o) t = 'o Event.event

let%server raw_conv
  : (url * 'a fragment, url * 'a) converter
  = Conv.pair Conv.url Conv.fragment

let%server serialize t =
  raw_conv.serialize (t.url, t.conv.deserialize)
let%client deserialize s =
  let url, deserial_msg =
    ~%raw_conv.deserialize s
  in
  let event, send = Event.create () in
  let handler msg = send (deserial_msg msg) in
  Broadcast.subscribe url handler ;
  event

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

Figure 8: `BroadcastEvent`: Shared reactive events

and produces a value of type `t`. Here, we use a converter explicitly in order to transfer elements on the broadcast bus.

The implementation of the API is shown in Figure 8. On the server, a `BroadcastEvent.t` is composed of a converter that is used to transfer elements together with a URL. The `create` function starts by creating an untyped broadcast endpoint. We then use `Event.iter` to serialize and then send each occurrence of the provided event.

We now need to create a converter for `BroadcastEvent.t`. We need to transmit two values: the URL of the broadcast endpoint, so that the client can subscribe, and the deserialization part of the provided converter, so that the client can decode the broadcasted messages. `raw_conv` provides a converter for a pair of a URL and a fragment. In addition to receiving this information, the client deserializer creates a new event stream and subscribes to the broadcast endpoint. We connect the broadcast output to the event stream by passing along all the (deserialized) messages.

As we can see in this example, we can use converters explicitly to setup very sophisticated communication schemes in a safe and typed manner. We also use the client deserialization step to execute stateful operations as needed on the client. Note that using a converter here allows effective use of resources: the only clients that subscribe to the broadcast are the ones that really need the event stream, since it has been injected.

3.3 Heterogeneous implementations

Shared sections make it possible to write code for the client and the server at the same time. This provides a convenient way of writing terse shared implementations, without duplicating logic and code. This does not necessarily entail that everything is shared. In particular, base primitives might differ between client and server, though the overall logic is the same. Just as we can implement heterogeneous datatypes with different client- and server-side representations, we can also provide interfaces common to the client and the server, with different client- and server-side implementations. We consider the case of database access. We first assume the existence of a server function `get_age` of type `string -> int` that performs a database query and returns the age of a person.

We can easily create a client version of that function via our RPC API of Figure 1.

```
let%server get_age_rpc = Rpc.create get_age
let%client get_age = %get_age_rpc
```

The API is then:

```
val%shared get_age : string -> int
```

We can use this function to write widgets that can be used either on the client or on the server:

```
let%shared personwidget name =
  div ~a:[a_class "person"] [
    text (name^" : "^string_of_int(get_age name))
  ]
```

This technique is used pervasively in Eliom to expose implementations than can be used either on the client or on the server with similar semantics, in a very concise way.

3.4 Mixed client-server data structures

We can readily embed client fragments inside server data structures. Having explicit location annotations really helps here. It would not be possible to achieve this for arbitrary data structures if the client-server delimitations were implicit.

As a first example of such a mixed data structure, consider a list of button names (standard server-side strings) and their corresponding client-side actions. Here is a function that takes such a list and builds an unordered HTML list of buttons.

```
let%server button_list
  (lst : (string * handler fragment) list) =
  ul (List.map (fun (name, action) ->
      li [button
            ~button_type:`Button
            ~a:[a_onclick action]
            [text name]])
      lst)
```

3.4.1 HTML. A common idiom in Web programming is to generate the skeleton of a Web page on the server, then fill in the holes on the client with dynamic content, or bind dynamic client-side behaviors on HTML elements. In order to do that, the usual technique is to use the `id` or `class` HTML properties to identify elements, and to manually make sure that these identifiers are used in a coherent manner on the client and the server.

Eliom simplifies this process by mean of a client-server HTML library that allows injections of HTML elements to the client. Figure 9 shows a simplified API, which is uniform across clients and servers. The API provides combinators such as the div function shown, which builds a div element with the provided attributes and child elements. We already used this HTML API in several previous examples.

```
type%shared attribute
type%shared element

val%shared div :
  ?a:(attributes list) -> element list -> element

val%server a_onclick :
  (Event.t -> bool) fragment -> attribute

module%server Client : sig
  val node : element fragment -> element
end
```

Figure 9: The simplified HTML API

On the server, HTML is implemented as a regular OCaml datatype. When sending the initial HTML document, this datatype is converted to a textual representation. This ensures compatibility with JavaScript-less clients and preserves the usual behavior of a Web server.

On the client, we represent HTML nodes directly as DOM trees. The mismatch between client and server implementations does not preclude us from providing a uniform API. However, to permit injections of HTML nodes from the server to the client, special care must be taken. In particular, we equip each injected node with an id, and id is the only piece of data sent by the serialization function. The deserialization function then finds the element with the appropriate id in the page. The a_onclick function finds the appropriate HTML element on the client and attaches the specified handler.

The fact that we use a uniform API allows us to abstract the specificities of the DOM and to provide other kinds of representations, such as a virtual DOM approach. A further improvement that fits in our design is nesting client HTML elements inside server HTML documents without any explicit DOM manipulation. This is done by the Client.node function (Figure 10), which takes a client fragment defining an HTML node and converts it to a server-side HTML node that can be embedded into the page. This function works by including a placeholder element server-side. The placeholder is later replaced by the actual element on the client.

```
let%server node (x: element fragment) : element =
  let placeholder = span [] in
  let _ = [%client
    let placeholder = ~%placeholder in
    let node = ~%x in
    Option.iter
      (Dom.parent placeholder)
      (fun parent ->
        Dom.replaceChild parent placeholder node)
  ] in
  placeholder
```

Figure 10: Implementation of Client.node

3.5 Shared values

The Eliom features we have described allow us to encode *shared values*, that is, values that have meaning both on the client and the server. The API is described in Figure 11 while the implementation is shown in Figure 12. Implementation of the converter for shared values is shown in Figure 13.

```
type%server ('a, 'b) shared_value =
  { srv : 'a ; cli : 'b fragment }
type%client ('a, 'b) shared_value = 'b

val%server local : ('a, 'b) shared_value -> 'a
val%client local : ('a, 'b) shared_value -> 'b

val%server cli : ('a, 'b) shared_value -> 'b fragment
val%client cli : ('a, 'b) shared_value -> 'b
```

Figure 11: Shared values API

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

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

Figure 12: Shared values Implementation

```
let%server shared_conv
  : (('a, 'b) shared_value,
     ('a, 'b) shared_value) converter = {
  serialize =
    (fun x -> Conv.fragment.serialize x.cli);
  deserialize = Conv.fragment.deserialize
}
```

Figure 13: Converter for shared values

The server-side implementation of a shared value clearly needs to contain a fragment that can be injected on the client. On the other hand, the client cannot possibly inject a value on the server, so the client-side representation only consists of a fragment. For injecting a server-side shared value on the client, we use a converter whose server-side portion serializes only the fragment, and whose client-side portion deserializes this fragment.

Shared values are very useful when a given operation needs to be performed both on the server and on the client, but in a way that matches the specific requirement of each side. As an example, we present a *shared (association) table* API for storing data of interest on both the server and the client. We use strings as keys. The type of tables ('a, 'b)table contains two type variables 'a and 'b, corresponding to the server- and client-side contents respectively. For the sake of simplicity, we assume that the type variables 'a and 'b must be instantiated such that there exists an appropriate converter. The API in Figure 14 provides add and find operations, as is typical for association tables, which are available on both sides.

Our goal is to have a table API well-adapted for Eliom's client-server style of programming. On the server, the table is to be used while serving a request, e.g., for locally caching data obtained from complex database queries. It is frequently the case that the client needs access to the same data; in that case, it is desirable that we avoid performing multiple RPCs. To achieve this, the semantics of the server-side addition operation (function add) is such that the value does not only become available for future server-side lookups, but also for client-side lookups. Of course, additional items may be added client-side, but then there is no expectation of server-side addition; the server-side table may not even exist any longer, given that it was local to the code handling the request.

The implementation is shown in Figure 15. A table is implemented as a pair of a server-side string-indexed hash table and a client-side one. The server-side add implementation stores a new value locally in the expected way, but additionally builds a fragment that has the side-effect of performing a client-side addition. The retrieval operation (`find`) is simple on both sides; we just look up a key on the local table.

Going further, shared values empower an approach to reactive programming that is well-adapted for Eliom's client-server paradigm (Shared reactive programming 2017). This approach is the subject of ongoing work.

```
type%shared ('a, 'b) table

val%server add :
  ('a, 'b) table -> string -> 'a -> unit
val%client add :
  ('a, 'b) table -> string -> 'b -> unit

val%server find : ('a, 'b) table -> string -> 'a
val%client find : ('a, 'b) table -> string -> 'b
```

Figure 14: SharedTable API

```
type%shared ('a, 'b) table =
  ((string, 'a) Hashtbl.t,
   (string, 'b) Hashtbl.t) shared_value

let%server add tbl id v =
  let server_tbl = local tbl
  and client_tbl = cli tbl in
  let _ =
    [%client Hashtbl.add ~%client_tbl ~%id ~%v ]
  in Hashtbl.add server_tbl id v

let%client add = Hashtbl.add

let%shared find tbl id =
  Hashtbl.find (local tbl) id
```

Figure 15: SharedTable Implementation

3.6 A sophisticated example: accordions

We now demonstrate how it is possible to implement the well-known widget *accordion*. An accordion is a kind of application menu that displays collapsible sections in order to present information in a limited amount of space. The section titles are always visible. The content of a section is shown when the user clicks on its title. Only one section is open at a time.

In our example, sections are implemented independently and attached to the accordion given as parameter. The distinctive characteristic of our implementation, made possible by the two-level language, is that a section can be generated freely either on the server or on the client, and attached to an existing accordion. The example contains three sections, two generated server-side, the other added dynamically client-side to the same accordion.

The code is shown in Figure 16. The data structure representing the accordion contains only a reference to a client-side function that closes the currently open section. Functions `new_accordion` and `accordion_section` are included in both the server and client programs (shared sections). Function `switch_visibility` is implemented client-side only. It just adds or removes an HTML class to the element, which has the effect of hiding or showing the element through CSS rules. Function `my_accordion` builds a server-side HTML page containing an accordion with two sections. It also sends to the client process, together with the page, the request to

```
let%client switch_visibility (elt : Html.elt) =
  if Class.contain elt "hidden"
  then Class.remove elt "hidden"
  else Class.add elt "hidden"

type%shared toggle = (unit -> unit) ref fragment

let%shared new_accordion () : toggle =
  [%client ref (fun () -> ()) ]

let%shared accordion_section
    (accordion : toggle) s1 s2 : Html.elt =
  let contents =
    div ~a:[a_class ["contents"; "hidden"]]
      [text s2]
  in
  let handler = [%client fun _ ->
    ! ~%accordion(); (*close previous section*)
    ~%accordion :=
      (fun () -> switch_visibility ~%contents);
    switch_visibility ~%contents
  ]
  in
  let title =
    div ~a:[a_class ["title"]; a_onclick handler]
      [text s1]
  in
  div ~a:[a_class ["section"]] [title; contents]

let%server my_accordion () : Html.elt =
  let accordion = new_accordion () in
  div [
    accordion_section
      accordion
      "Item 1" "Server side generated" ;
    Client.node [%client
      accordion_section
        ~%accordion
        "Item 2" "Client side generated"
      ] ;
    accordion_section
      accordion
      "Item 3" "Server side generated" ;
  ]
```

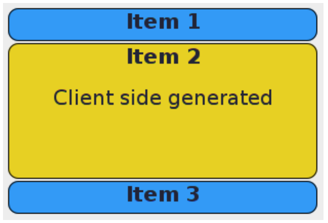

Figure 16: The accordion widget

create the accordion (client fragment in function `new_accordion`) and to append a new section to the accordion. For this purpose, we use function `Client.node` on line 36.

3.7 Going further

Our examples demonstrate how the combination of fragments, injections and converters can be used to build rich Web development libraries that provide convenient programming interfaces. Using these same building blocks, the Eliom library additionally implements uni- and bi-directional channels, progressive fetching of data, correct-by-construction links, and client-server reactive programming. Interestingly, a common pattern arising across these examples (just like for our RPC and HTML examples of Sections 3.2.1 and 3.4.1) is relating server and client datatypes that differ in their structure and APIs, but that have related intuitive meaning. Of course, the same building blocks and patterns can be used by the programmer to implement additional components outside the Eliom library, thus catering for their specific use cases.

4 IMPLEMENTATION

Eliom is implemented as an extension of the OCaml programming language. In this section, we detail how this extension is implemented, describing both the typechecking part and the semantics part.

4.1 Global overview

Before detailing each part of the implementation, we give a global overview of the Eliom compilation pipeline. A schema is shown in Figure 17.

We modified the OCaml typechecker to handle Eliom constructs. Our modified typechecker is presented in Section 4.2. Once the Eliom code has been typechecked, we use it to generate two OCaml files, the client part and the server part, through the slicing method presented in Section 4.3. The slicing relies on inserting primitives that implement the communication mechanism of Eliom (as described in Section 3.1.1); we present the semantics of said primitives in Section 4.4. After slicing, the two generated files are pure OCaml code and can be compiled with the regular OCaml compiler. The client code is finally translated to JavaScript using the js_of_ocaml compiler. In order to integrate gracefully with the OCaml ecosystem, Eliom is provided as a PPX syntax extension, which is OCaml's standard syntax extension mechanism.

This workflow ensures several desirable properties:

- Eliom is fully compatible with OCaml. Thus, Eliom code can be linked against standard OCaml code. The various OCaml tools are compatible with Eliom code.
- The generated code is typechecked again by the vanilla OCaml typechecker, which increases trust.
- The behavior of the language extension is predictable: Eliom code that contains neither fragments nor injections is copied straight to the generated OCaml files, and behaves exactly like it would in a regular OCaml program.

4.2 Interaction with the OCaml type checker

The OCaml typechecker modifications have been kept rather small, amounting to a patch of less than a hundred lines changed over fifteen files. This is important both for review and to ease updates to future OCaml versions.

The modifications follow closely the formalization in (Radanne et al. 2016): symbols (such as variables, types or module names) are now equipped with a side that is either client or server. The lookup mechanism has been modified to respect the symbol side, and the current side is kept track of during typechecking. The modified typechecker tracks the side across the client/server boundary.

An important point is that the OCaml language is obviously much larger than our formalization of Eliom. This issue is mitigated by the fact that Eliom typing rules only differ from regular ML on the boundaries between client and server. In particular, given that sections are only allowed at toplevel, a section containing neither fragments nor injections can be handled by the vanilla OCaml typechecker. This means that we don't need special handling for functors and the object system. By being conservative in the type conversions across client-server boundaries (in particular, by prohibiting existential and universal types), we avoid difficulties related to complex features of the OCaml type system, such as GADTs and first-class modules.

4.3 Client-server code separation

We present how Eliom code is split into server and client code. To simplify this presentation, we assume that injections are always variables. Programs where injections contain expressions can be transformed to respect this constraint by hoisting the expression above the enclosing fragment or client section.

The needed primitives are shown in Figure 19. We use specific types for the identifiers (`closure_id` and `inj_id`). For the sake of exposition, we simply represent identifiers as strings in the examples. The actual implementation uses various means to ensure that identifiers do not collide.

Note that we use low-level communication primitives which cannot be made type-safe. We can however annotate the generated OCaml code with the type information inferred by the Eliom typechecker, which ensures that these primitives are used in a safe way. Sections 4.3.1 and 4.3.2 describe slicing of client and server sections, respectively.

4.3.1 Client sections. Client sections can only contain injections that are statically known. Indeed, since sections may only occur at toplevel, injections can only refer to toplevel server identifiers. Thus, in the server code, we can replace each client section by a sequence of calls to the server primitive `push_injection`, where each call registers a value to be sent to the client. In the client code, each injection is replaced by a call to the client primitive `get_injection` which returns the value of the injection. Figure 18 provides an example.

4.3.2 Server sections. As opposed to injections in client sections, fragments that are created during the execution of a server section are not statically known. Besides, several instances of a same fragment might be created with different injected values. Consider for instance the following piece of code:

```
let%server i = [%client ~%(Random.int 10) + 1]
```

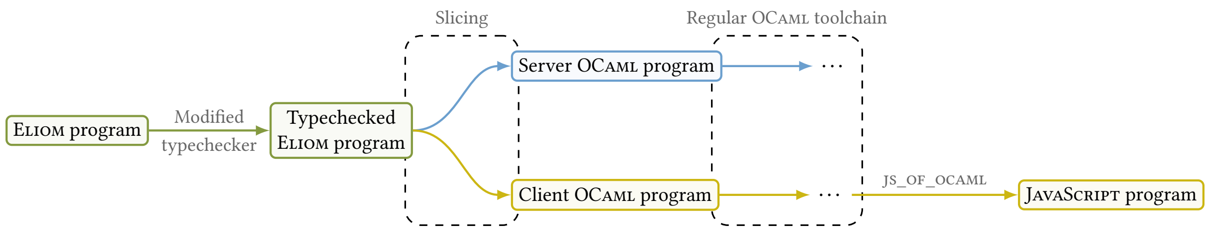

Figure 17: The Eliom compilation pipeline

	Server	Client
`let%client c = ~%s + 1`	`let () = push_injection "A.s1" s`	`let c = get_injection "A.s1" + 1`

`let%server y =`
`  [%client 2 + ~%x ]`
	`let y = fragment "B1" (x)`
	
	`let () = push_fragments "B"`
		`let () =`
		`  register_closure "B1" (fun x -> 2 + x)`
		`let () = execute_fragments "B"`

Figure 18: Client-server code separation

Given these constraints, client fragments are processed in two steps. First, we extract the closure corresponding to the fragment parameterized over its injections. This closure is given a unique identifier and is registered client-side through the primitive `register_closure`. Then, we replace each occurrence of a fragment by a call to `fragment` with the closure identifier and the various injections passed as arguments. On the client, the primitive `execute_fragments` then evaluates the fragments that have been created server-side in this section. We show an example in Figure 18.

4.4 Semantics of primitives

As detailed in Section 3.1.1, execution of client and server code is not synchronized. On the contrary, server code is executed first, injections are sent to the client, then client code is executed. Still, the order of evaluation between client code and fragments is maintained.

To implement this semantics, we use a queue to store the fragments that the client will need to execute. A schema of an execution is shown in Figure 20. In the server code, each call to `fragment` generates a fresh identifier and registers a new fragment to be executed. In each section, a call to `push_fragments` pushes those fragments in the queue. The queue is then sent to the client. In the client code, the `execute_fragments` primitive dequeues the fragments associated to the corresponding section and evaluates them. In order to evaluate them, it uses the closure registered by `register_closure`. Finally, the value of the fragment is stored.

```
val%server fragment :
  closure_id -> 'injs -> 'a fragment
val%server push_fragments : id -> unit
val%server push_injection : inj_id -> 'a -> unit

val%client get_injection : inj_id -> 'a
val%client register_closure :
  closure_id -> ('a -> 'b) -> unit
val%client execute_fragments : id -> unit
```

Figure 19: Primitives

This execution scheme ensures that the evaluation order of client code is preserved.

Injections inside client sections follow a similar scheme. However injections are values, and thus do not need to be evaluated. Hence, they can be simply stored into a map. We use `get_injection` to retrieve values from the map. When the injection is a client fragment, we also access the fragment table in order to retrieve the value.

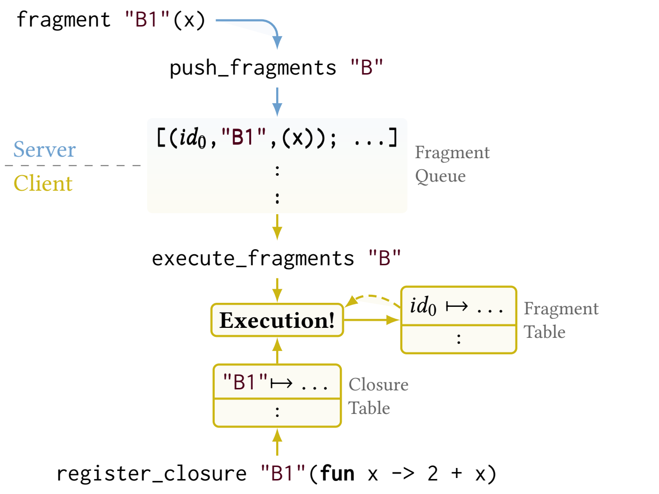

Figure 20: Schema of the semantics of fragments

4.5 Converters

In our formalization (Radanne et al. 2016), converters need to be explicitly provided for each injection. This is rather inconvenient. To overcome this, our current implementation performs serialization and deserialization in a general way using a modified version of the `Marshal` module from the standard OCaml distribution. While in general marshalling is neither type nor memory safe, Eliom typing ensures proper usage. Furthermore, only the client process deserializes data through OCaml's standard marshalling mechanism. Therefore, the server can never be exposed to a deserialization error, nor be coerced into deserializing malicious contents.

Despite using the general marshalling machinery, current Eliom still allows for custom converters. This works as follows. Instead of being provided explicitly when doing an injection (which would involve syntactic overhead), custom converters can be attached directly to values. Our marshalling implementation uses these custom converters whenever available.

One potential solution for type-specific converters without explicitly annotating injections is to use ad-hoc polymorphism, such as type-classes or implicits (White et al. 2014). Converters can then be inferred. We plan to use the latter when they become available in OCaml.

In the presence of converters, type-checking injections is straightforward. Suppose we inject an element of server-side type α with a converter (α, β) `converter`. The injected value necessarily belongs to the client-side type β. In the case of implicit converters (White et al. 2014), if no converter is known, we reject the injection.

5 RELATED WORK

5.1 Unified client-server languages

Various directions have been explored to simplify Web development and to adapt it to current needs. Eliom places itself in one of these directions, which is to use the same language on the server and the client. Several unified client-server languages have been proposed. They can be split in two categories depending on their usage of JavaScript. JavaScript can either be used on the server, with Node.js, or as a compilation target, for example with Google Web Toolkit for Java or Emscripten for C.

The approach of compiling to JavaScript was also used to develop new client languages aiming to address the shortcomings of JavaScript. Some of them are new languages, such as Haxe, Elm or Dart. Others are only JavaScript extensions, such as TypeScript or CoffeeScript. These various proposals do not help in solving client-server communication issues: the programmer still writes the client and server code separately and must ensure that messages are written and read in a coherent way.

5.2 Tierless languages and libraries

Several other languages share with Eliom the characteristic of mixing client and server code in an almost transparent way. In this section, we attempt to create a taxonomy of tierless programming languages. We first give a high-level comparison of the various trade-offs involved.

Inference of code location. In Eliom, code location is specified through manual annotations. In several other approaches, code location is inferred based on known elements (database access is on the server, dynamic DOM interaction is done on the client, etc). Such inference can be done either in a type-directed manner (Cooper et al. 2006) or via a global control flow analysis (Chong et al. 2007; Opa 2017; Philips et al. 2014). Another approach is to operate code slicing at runtime. This is mostly used by JavaScript-based systems. These various approaches present a different set of compromises:

- While very elegant, typed-directed inference of code location is difficult to integrate into an existing language, as it would mean profound changes in the language's type system. Furthermore, type and effect systems, such as the one in Links, are still an active area of research.
- A global control flow analysis prevents separate compilation. Also, given the interaction between the control flow analysis and other code transformations, it can be difficult to know where each piece of code is executed (as pointed out in the last section of Chlipala (2015b)).
- Good inference of code location is difficult to achieve within an effectful language.
- Inference cannot be as precise as explicit annotations. For example, it does not work if the program builds data structures that mix client fragments and other data, as in Section 3.4.
- We believe that the efficiency of a complex Web application relies a lot on the programmer's ability to know exactly where the computation is going to happen at each point in time. In many cases, both choices are possible, but the result is very different from a user or a security point of view.

Eliom has separate type universes for client and server types (see Section 3.2.2). This allows the type system to check which functions and types are usable on which side. We believe that manual annotation combined with type-level tracking of code location provides the best compromise between correctness, expressivity, and the ability to implement Eliom as an extension of an existing language.

Runtime communications. Eliom uses asymmetric communication between client and server (see Section 3.1.1). Everything needed to execute the client code is sent during the initial communication that also sends the Web page (unless RPC, as presented in Section 3.2.1, is used).

Various other communication schemes have been proposed. Most other languages that provide static code slicing only allow dynamic communication. On the other hand, some programming languages (such as Hop) provide dynamic slicing at run time. The combination of compile-time code slicing and asymmetric communication is a novel feature of Eliom.

Details on some specific approaches. We now provide an in-depth comparison with the most relevant approaches.

Ur/Web (Chlipala 2015a,b) is a new statically typed language especially designed for Web programming. While similar in scope to Eliom, it follows a very different approach: Ur/Web uses whole-program compilation and a global control flow analysis to track locations. This makes some examples hard to express, such as the one in Section 3.4. Client and server locations are not tracked by the type system and are not immediately visible in the source code, which can make compiler errors hard to understand, and is incompatible with separate compilation. Furthermore, contrary to Eliom, several primitives such as RPC are hardcoded in the language, which makes it less easy to extend with libraries providing new client-server behaviors.

Hop (Boudol et al. 2012; Serrano and Queinnec 2010) is a dialect of Scheme for programming Web applications. Its successor, Hop.js (Serrano and Prunet 2016), takes the same concepts and brings them to JavaScript. It uses location annotations similarly to Eliom and provide facilities to write complex client-server applications. However, as a Scheme-based language, it does not provide static typing. Slicing is performed at runtime. In particular, contrary to Eliom, Hop does not statically enforce the separation of client and server universes (such as using database code inside the client).

Links (Cooper et al. 2006) is an experimental functional language for client-server Web programming with a syntax close to JavaScript and an ML-like type system. Its type system is extended with a notion of *effects*, allowing a clean integration of database queries in the language. It does not provide any mechanism to separate client and server code, so they are shared by default, but it uses effects to avoid erroneous uses of client code in server contexts (and conversely). Compared to Eliom, compilation is not completely available and Links does not provide an efficient communication mechanism.

Haste (Ekblad and Claessen 2014) is an extension of Haskell similar to Eliom. Instead of using syntactic annotations, it embeds client and server code into monads. This approach works well in the Haskell ecosystem. However Haste makes the strong assumption that there exists a universe containing both client and server types, shared by the client and the server. Eliom, on the contrary, does not make this assumption, so the monadic `bind` operator for client fragments, of type `('a -> { 'b })-> { 'a } -> { 'b }`, makes no sense: `'a` would be a type both on the server and on the client, which is not generally true. Haste uses type-directed static slicing but only provides dynamic communication.

Meteor.js (Meteor.js 2017) is a framework where both the client and the server sides of an application are written in JavaScript. It has no built-in mechanism for sections and fragments but relies on conditional `if` statements on the `Meteor.isClient` and `Meteor.isServer` constants. It does not perform any slicing. This means that there are no static guarantees over the respective execution of server and client code. Besides, it provides no facilities for client-server communication such as fragments and injections. Also, compared to Eliom, this solution only provides coarse-grained composition.

5.3 Staged meta-programming

The type system and programming model of Eliom is very similar to the one provided by staged meta-programming. Eliom simply provides only two stages: stage 0 is the server, stage 1 is the client. Eliom's client fragments are the equivalent of stage quotations. There has been a lot of work in staged meta-programming and partial evaluation. We only mention two relevant works.

MetaOCaml (Kiselyov 2014) is an extension of OCaml for meta programming. It introduces a quotation annotation for staged expressions, whose execution is delayed. The main difference is the choice of universes: Eliom has two universes, client and server, which are distinct. MetaOCaml has a series of universes, for each stage, sequentially included in one another.

Feltman et al. (2016) presents a slicing technique for a two-staged simply typed lambda calculus. Their technique is similar to the one used in Eliom. One difference is the lack of cross-staged persistency (which is solved in Eliom using converters).

5.4 Distributed programming

Eliom is related to the notion of distributed programming. Usually, distributed programming involves several actors and back-and-forth communication. The actors may or may not have the same capabilities. Client-server can be seen as a degenerate case where there are only two actors, one privileged communication direction, and very asymmetric capabilities.

ML5 (VII et al. 2007) is an ML language that introduces new constructs for type-safe communication between distributed actors. It is geared towards a situation where all actors have similar capabilities. It uses dynamic communication, which makes the execution model very different from Eliom. The constructs introduced by ML5 could be used in Eliom to distribute tasks across several servers.

6 CONCLUSION

We have described how to use Eliom, a programming language for client-server Web applications, to create new applications and libraries. We presented several new programming patterns that are enabled by the new language constructs provided by Eliom. We justified these programming patterns by presenting numerous examples extracted from the Eliom library that illustrate standard website features. Finally, we presented the current implementation of Eliom.

The core Eliom language extension has been formalized in Radanne et al. (2016). This language extension is sufficiently small to be reasoned about and implemented on top of an existing language, such as OCaml. It is also expressive enough to allow the implementation, without any additional language built-in constructs, of all kinds of widgets and libraries for Web programming.

The implementation of Eliom as an extension of an existing language makes it possible to reuse a large set of existing libraries and to benefit from an already large community of users. This is crucial because Web programming is never about the Web per se, but almost always related to other fields for which dedicated libraries are necessary.

Explicit annotations are used to indicate the location where program execution takes place. Adding them is really easy for programmers and is a good way to help them see exactly where computation is going to happen, which is crucial when developing real-size applications. Eliom makes it impossible to introduce unwanted communication by mistake.

Eliom makes strong use of static typing to guarantee many properties of the program at compile time. Developing the client and server parts as a single program allows to guarantee the consistency between the two parts, and statically check all communications (*e.g.*, injections, server push, and remote procedure calls).

These design choices have always been guided by concrete uses. From the beginning, Ocsigen has been used for developing real-scale applications. The experience of users has shown that the use of a tierless language is more than a viable alternative to the traditional

Web development techniques, and is well suited to the current evolution of the Web into an application platform. The fluidity gained by using a tierless programming style with static typing matches the need of a new style of applications, combining both the advantages of sophisticated user interfaces and the specificities of Web sites (connectivity, traditional Web interaction, with URLs, back button, ...). This is made even more convenient through the use of features such as an advanced service identification model and the integration of reactive functional programming that are provided by ELIOM but have not been covered here.

REFERENCES

Vincent Balat. 2013. Client-server web applications widgets. In *WWW'13 dev track: Proceedings of the 22nd international conference on World Wide Web*. Rio de Janeiro, Brazil, 19–22. http://dl.acm.org/citation.cfm?id=2487788.2487795

Vincent Balat, Pierre Chambart, and Grégoire Henry. 2012. Client-server Web applications with Ocsigen. In *WWW'12 dev track: Proceedings of the 21nd international conference on World Wide Web*. Lyon, France, 59. http://hal.archives-ouvertes.fr/hal-00691710

Vincent Balat, Jérôme Vouillon, and Boris Yakobowski. 2009. Experience report: Ocsigen, a Web programming framework. In *ICFP*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 311–316.

Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano. 2012. Reasoning about Web Applications: An Operational Semantics for HOP. *ACM Trans. Program. Lang. Syst.* 34, 2 (2012), 10.

Adam Chlipala. 2015a. An Optimizing Compiler for a Purely Functional Web-Application Language. In *ICFP*.

Adam Chlipala. 2015b. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 153–165. DOI: http://dx.doi.org/10.1145/2676726.2677004

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 31–44. DOI: http://dx.doi.org/10.1145/1294261.1294265

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *FMCO*. 266–296.

Anton Ekblad and Koen Claessen. 2014. A Seamless, Client-centric Programming Model for Type Safe Web Applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 79–89. DOI: http://dx.doi.org/10.1145/2633357.2633367

Eliom 2017. *Eliom web site.* https://ocsigen.org/eliom.

Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. 2016. Automatically Splitting a Two-Stage Lambda Calculus. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 255–281. DOI: http://dx.doi.org/10.1007/978-3-662-49498-1_11

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 86–102. DOI: http://dx.doi.org/10.1007/978-3-319-07151-0_6

Meteor.js 2017. *Meteor.js.* http://meteor.com.

Ocsigen Toolkit 2017. *Ocsigen Toolkit.* http://ocsigen.org/ocsigen-toolkit/.

Opa 2017. *Opa web site.* http://opalang.org/.

Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development Without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 69–81. DOI: http://dx.doi.org/10.1145/2661136.2661146

Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A Core ML Language for Tierless Web Programming. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.), Vol. 10017. 377–397. DOI: http://dx.doi.org/10.1007/978-3-319-47958-3_20

React 2017. *React.* http://erratique.ch/software/react.

Gabriel Scherer and Jérôme Vouillon. 2010. Macaque : Interrogation sûre et flexible de base de données depuis OCaml. In *Ving et unième journées francophones des langages applicatifs (Studia Informatica Universalis)*. Hermann, La Ciotat, France, –. https://hal.archives-ouvertes.fr/hal-00495977

Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 180–192. DOI: http://dx.doi.org/10.1145/2951913.2951916

Manuel Serrano and Christian Queinnec. 2010. A multi-tier semantics for Hop. *Higher-Order and Symbolic Computation* 23, 4 (2010), 409–431.

Shared reactive programming 2017. *Shared React.* https://ocsigen.org/eliom/5.0/manual/clientserver-react.

Tutorial 2017. *Ocsigen Tutorial.* https://ocsigen.org/tuto/manual.

TyXML 2017. *TyXML.* http://ocsigen.org/tyxml/.

Tom Murphy VII, Karl Crary, and Robert Harper. 2007. Type-Safe Distributed Programming with ML5. In *TGC (Lecture Notes in Computer Science)*, Gilles Barthe and Cédric Fournet (Eds.), Vol. 4912. Springer, 108–123.

Jérôme Vouillon. 2008. Lwt: a cooperative thread library. In *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, Eijiro Sumii (Ed.). ACM, 3–12. DOI: http://dx.doi.org/10.1145/1411304.1411307

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972. DOI: http://dx.doi.org/10.1002/spe.2187

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular Implicits. *ML workshop* (2014). http://www.lpw25.net/ml2014.pdf