Eliom: tierless Web programming from the ground up*

Gabriel Radanne
Univ Paris Diderot, Sorbonne Paris Cité, IRIF UMR 8243 CNRS
gabriel.radanne@pps.univ-paris-diderot.fr

Jérôme Vouillon
CNRS, IRIF UMR 8243, Univ Paris Diderot, Sorbonne Paris Cité, BeSport
jerome.vouillon@pps.univ-paris-diderot.fr

Vincent Balat
Univ Paris Diderot, Sorbonne Paris Cité, IRIF UMR 8243 CNRS, BeSport
vincent.balat@univ-paris-diderot.fr

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

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 distributed 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 and 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 introduces a new paradigm for web programming, and how this paradigm allows building complex libraries easily, safely, and in a composable manner.

Keywords Web, client-server, OCaml, ML, Eliom, Ocsigen, functional programming

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 it all 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 client and server (*e.g.*, notifications and messaging). Such communication is 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 unified languages that encompass both client-side and server-side code, for example Links (Cooper et al. 2006) and Ur/Web (Chlipala 2015a,b). These *tierless* languages allow better encapsulation and composition. When combined with static typing, they also allow statically checking client-server communication.

Eliom is an extension of OCaml that can express client- and server-side code in an integrated way. Eliom thus provides the composability advantages of tierless programming, namely composability and seamless type-safe client-server interaction, but also brings in the benefits of an existing language. The Eliom-specific primitives 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.).

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) and database interaction (Scherer and Vouillon 2010). Designing Eliom as a minimalist extension of OCaml aligns well with its position in the Ocsigen and wider OCaml ecosystems. Namely, we

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

[Copyright notice will appear here once 'preprint' option is removed.]

have implemented ELIOM by extending the OCAML compiler; but we have only modified the compiler frontend, with the backend and runtime remaining untouched. ELIOM code can thus use pre-existing and unmodified OCAML packages. This has allowed the OCSIGEN team to build a comprehensive set of Web development libraries that follow the tierless paradigm of ELIOM, without sacrificing compatibility with the wider OCAML ecosystem.

We focus here on the practical aspects of programming with ELIOM. More specifically, we discuss how our language enables a paradigm well-suited for easily and safely implementing libraries and applications that involve complex client-server interactions.

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 is able to manage several sessions at the same time, with the possibility of sharing data between sessions, and to keep 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. It persists until the browser tab is closed, or until the user follows an external link.

Composition The ELIOM languages 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 libraries and building blocks 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) 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 et al. 2012; Balat 2013). This choice is motivated by the fact that we believe that the programmer must be well aware of where the code is to be executed, to avoid unnecessary remote interactions. 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 constant 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 that a declaration is performed on the server, or on the client:

```
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**.

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.

```
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.

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.

Injection also enable us to access client fragments which have 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 on how to use the ELIOM extensions. Our two examples are extracted from code that appears in the OCSIGEN tutorial and in the ELIOM library

(Eliom). Both examples are slightly idealized for clarity of exposition.

3.1 User interface widget

We can define a button that increments a client-side counter and invokes a callback each time it is clicked. We use a DSL to specify HTML documents. The callback `action` is a client function. The state is stored in a client-side reference. The `onclick` button callback is a client function that modifies the reference, and then calls `action`. This illustrates that one can define a function that builds on the server a Web page fragment with a client-side state and a parameterized client-side behavior. It would be 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) ]]
    [pcdata "Increment"]
```

`counter` is a server widget that 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 -> int) fragment -> Html.t
```

This widget is easily composable: the client state included cannot affect another widget and it can be used to build bigger widgets. Furthermore, the execution is efficient, given that the server only ever sends data along with the initial version of the page

3.2 Remote procedure call library

When using fragments and injections, the only communication between client and server that takes place is the original HTTP request and response. However, further communication is sometimes desirable. A remote procedure call (RPC) 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.

In the example, we first create server-side an RPC endpoint using the function `Rpc.create`. This function returns a value of type `(int, int)Rpc.t`, that is a RPC with argument and return value both of type `int`. `Rpc.t` is an abstract type on the server, but expands to a function type on the client. The input of the function is transmitted from the client to the server, which is why we require a `Json.decoder` when creating an endpoint. This decoder parses the untrusted value serialized as JSON by the client.

We give a quick sketch of how this API is implemented. Let us first assume we have a function `serve` of type `string -> (request -> answer)-> unit` that

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

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

Figure 1. The simplified RPC api

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

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

Figure 2. An example using the RPC api

can create 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.

When doing an injection, instead of transmitting the endpoint, we simply send the URL of the endpoint, and nothing else. The client side 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.

The ability to transform the data before sending it to the client is made possible by the use of *converters* (Radanne et al.).

Conclusion

We provided a brief introduction to the ELIOM language and discussed the programming paradigm that it enables. In the talk, we will present additional, more complex examples, various other libraries along with their implementation, and an in-depth comparison to other tierless languages.

References

V. Balat. Client-server Web applications widgets. In *WWW'13 dev track*, 2013. ISBN 978-1-4503-2038-2.

V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a Web programming framework. In *ICFP*, pages 311–316. ACM, 2009. ISBN 978-1-60558-332-7.

V. Balat, P. Chambart, and G. Henry. Client-server Web applications with Ocsigen. In *WWW'12 dev track*, page 59, Lyon, France, Apr. 2012.

A. Chlipala. Ur/Web: A simple model for programming the Web. In *POPL*, 2015a. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677004.

A. Chlipala. An optimizing compiler for a purely functional Web-application language. In *ICFP*, 2015b.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, pages 266–296, 2006.

Eliom. *Eliom web site*. http://ocsigen.org/.

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

G. Radanne, J. Vouillon, and V. Balat. Eliom: A core ML language for Tierless Web programming. Submitted to APLAS

2016. URL https://hal.archives-ouvertes.fr/hal-01349774.

G. Scherer and J. Vouillon. Macaque : Interrogation sûre et flexible de base de données depuis OCaml. In *21ème journées francophones des langages applicatifs*, 2010.

Tyxml. *Tyxml*. http://ocsigen.org/tyxml/.

J. Vouillon. Lwt: a cooperative thread library. In *ACM Workshop on ML*, 2008.

J. Vouillon and V. Balat. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience*, 44(8): 951–972, 2014. ISSN 1097-024X. doi: 10.1002/spe.2187.