Eliom: A core ML language for tierless Web programming*

Gabriel Radanne[1], Jérôme Vouillon[2], and Vincent Balat[3]

[1] Univ Paris Diderot, Sorbonne Paris Cité, IRIF UMR 8243 CNRS
gabriel.radanne@pps.univ-paris-diderot.fr
[2] CNRS, IRIF UMR 8243, Univ Paris Diderot, Sorbonne Paris Cité, BeSport
jerome.vouillon@pps.univ-paris-diderot.fr
[3] Univ Paris Diderot, Sorbonne Paris Cité, IRIF UMR 8243 CNRS, BeSport
vincent.balat@univ-paris-diderot.fr

Abstract. Eliom is a dialect of OCaml for Web programming in which server and client pieces of code can be mixed in the same file using syntactic annotations. This allows 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. The Eliom language is both sufficiently small to be implemented on top of an existing language and sufficiently powerful to allow expressing many idioms of Web programming.
In this paper, we present a formalization of the core language of Eliom. We provide a type system, the execution model and a compilation scheme.

Keywords: Web, client-server, OCaml, ML, Eliom, functional

1 Introduction

Web programming usually relies on the orchestration of numerous languages and tools. Web pages are written in HTML and styled in CSS. Their dynamic behavior is controlled through client-side languages such as JavaScript or ActionScript. These pages are produced by a server which can be written in about any language: PHP, Ruby, C++ ... They are produced based on information stored in databases and retrieved using a query language such as SQL.

Programmers must not only master these tools, but also keep synchronized the numerous software artifacts involved in a Web site and make them communicate properly. The server must be able to interact with the database, then send the relevant information to the client. In turn, the client must be able to understand the received data, and interact back properly with the server.

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

These constraints makes Web programming tedious and prone to numerous errors, such as communication errors. This issue, present in the Web since its inception, has become even more relevant in modern Web applications.

Separation of client and server code also hinders composability, as related pieces of code, that build fragments of a Web page and that define the specific behavior of these fragments, typically have to be placed in different files.

1.1 The need for tierless languages

One goal of a modern client-server Web application framework should be to make it possible to build dynamic Web pages in a *composable* way. One should be able to define on the server a function that creates a fragment of a page together with its associated client-side behavior; this behavior might depend on the function parameters. From this point of view, a DSL for writing HTML documents and serialization libraries are two key ingredients to assemble page fragments and communicate between client and server, but are not enough to associate client behaviors to these page fragments in a composable way.

This is where so-called *tierless* languages come into play. Such languages unify the client and server part of the application in one language with seamless communication. For most of these languages, two parts are extracted from a single program: a part runs on the server while the other part is compiled to JavaScript and runs on the client.

1.2 Eliom

We present Eliom, an extension of OCaml for tierless programming that supports composable and typesafe client-server interactions. Eliom is part of the larger Ocsigen [12,4] project, which also includes the compiler js_of_ocaml [24], a Web server, and various related libraries to build client-server applications. Besides the language presented here, Eliom comes with a complete set of modules, for server and/or client side Web programming, such as RPCs; a functional reactive library for Web programming; a GUI toolkit [16]; a powerful session mechanism, to manage the server side state of a Web applications on a per-tab, per-browser, or per-user basis; an advanced *service identification mechanism* [2], providing a way to implement traditional Web interaction (well-specified URLs, back button, bookmarks, forms, ...) in few lines of code. The Ocsigen project started in 2004, as a research project, with the goal of building a complete framework, usable in the industry. Ocsigen already has several industrial users, most using the Eliom language we present in this paper: BeSport [5], NYU gencore [13], Pumgrana [19], ...

1.3 A core language for tierless web programming

All of the modules and libraries implemented in Ocsigen, and in particular in the Eliom framework, are implemented on top of a core language that allows to express all the features needed for tierless web programming.

Composition. Eliom encourages the building of independent and reusable components that can be assembled easily. It 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. It also makes it possible to define libraries and building blocks without explicit support from the language.

Leveraging the type system. Eliom introduces a novel type system that allows composition and modularity of client-server programs while preserving type-safety and abstraction. This ensures, via the type-system, that client code is not used inside server code (and conversely) and ensures the correctness of client-server communications.

Explicit communication. Communication between the server and the client in Eliom is always explicit. This allows the programmer to reason about where the program is executed and the resulting trade-offs. The programmers can ensure that some data stay on the client or on the server, or choose how much communication takes place and where computation is performed.

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 ML.

These four properties lead us to define a core language, Eliom$_\varepsilon$, that features all these characteristics, while being small enough to be reasoned about in a formal way. Having a small core language into which more complex constructs can be reduced has several advantages. First, it is sufficient to trust the core language. All the desirable properties about the core language will also be valid for the higher-level abstractions introduced later.

A minimal core language also makes the implementation on top of an existing language easier. In the case of Eliom, it allows us to implement our extension on top of the existing OCaml compiler with a reasonably small amount of changes. By extending an existing language, we gain the ability to use numerous preexisting libraries: cooperative multitasking [23], HTML manipulation [22] and database interaction [20] are all provided by libraries implemented in pure OCaml that we can leverage for Web programming, both server and client side.

We present the Eliom language from a programming point of view in Section 2. We then specify Eliom$_\varepsilon$, an extension of core ML featuring the key points of Eliom, which allows us to describe the type system and the execution model of Eliom in Section 3 and the compilation model in Section 4 and Section 5.

2 How to: client-server Web programming

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 connections and 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 together 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.

Eliom is using manual annotations to determine whether a piece of code is to be executed server or client side [3,1]. This choice of explicit annotations is motivated by the fact that we believe that the programmer must be well aware of where the code is executed, to avoid unnecessary remote interactions. This also avoids ambiguities in the semantics and allows for more flexibility.

In this section, we 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 write 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.

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.

2.4 Examples

We show how these language features can be used in a natural way to build HTML pages with dynamic behavior in a composable fashion. More detailed examples are available on the Ocsigen website [12].

Increment button. 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 references 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 parametrized client-side behavior. It would be straightforward to extend this example with a second button that decrements the counter (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"]
```

List of server side buttons with client side actions. We can readily embed client fragments inside server datastructures. Having explicit location annotations really helps here. It would not be possible to achieve this for arbitrary datastructures if the client-server delimitations were implicit.

For instance, one can build an HTML unordered list of buttons from a list composed of pairs of button names and their corresponding client-side actions.

```
let%server button_list (l : (string * handler fragment) list) =
  let aux (name, action) =
    li [button ~button_type:`Button ~a:[a_onclick action] [pcdata name]]
  in ul (List.map aux l)
```

2.5 Libraries

These examples show how to build reusable widgets that encapsulate both client and server behavior. They also show some of the libraries that are provided with Eliom. In contrast to many Web programming languages and frameworks, which provides built-in constructions, all these libraries have been implemented only with the primitives presented in this paper. Using fragments and injections along with converters, which are presented later, we can implement numerous libraries such as remote procedure calls, client-server HTML or reactive programming directly inside the language, without any compiler support.

2.6 Client-server communication

In the examples above, we showed complex patterns of interleaved client and server code, including passing client fragments to server functions, and subsequently to client code. This would be costly if the communication between client and server were done naively. Instead, a single communication takes place: from the server to the client, when the Web page is sent. This is made possible by the fact that client fragments are not executed immediately when encountered inside server code. The intuitive semantics is the following: client code is not executed right away; instead, it is registered for later execution, once the Web page has been sent to the client. Then all the client code is executed in the order it was encountered on the server. This intuitive semantics allows the programmer to reason about Eliom programs, especially in the presence of side effects, while still being unaware of the details of the compilation scheme.

3 A client-server language

We present Eliom$_\varepsilon$, an extension of core ML containing the key features of Eliom. It differs from Eliom as follows. Shared sections are not formalized, as they can be straightforwardly expanded out into a client and a server section by duplicating the code. Additionally, we do not model the interactive behavior of Web servers. Thus, Eliom$_\varepsilon$ programs compute a single Web page.

3.1 Syntax

In order to clearly distinguish server code from client code, we use subscripts to indicate the location where a piece of syntax belongs: a 's' subscript denotes server code, while a 'c' subscript denotes client code. For instance, e_s is a server expression and τ_c is a client type. We also use 'ς' subscripts for expressions which are location-agnostic: they can stand for either s or c. When the location is clear from the context, we omit the subscripts.

The syntax is presented in Figure 1. It follows the ML syntax, with two additional constructs for client fragments and injections respectively. [26] was used as a base for the elaboration of Eliom$_\varepsilon$. The language is parametrized by its constants. There are different sets of constants for the server and for the client: $Const_s$ and $Const_c$. A program is a series of bindings, either client or server ones, ending by a *client* expression. The value of this expression will typically be the Web page rendered on the client's browser.

$$
\begin{array}{rll}
p ::= \texttt{let}_s\ x = e_s\ \texttt{in}\ p \mid \texttt{let}_c\ x = e_c\ \texttt{in}\ p \mid e_c & & \text{(Programs)} \\
e_s ::= c_s \mid x \mid \texttt{Y} \mid (e_s\ e_s) \mid \lambda x.e_s \mid \{\!\{\ e_c\ \}\!\} & & \text{(Expressions)} \\
e_c ::= c_c \mid x \mid \texttt{Y} \mid (e_c\ e_c) \mid \lambda x.e_c \mid f\%e_s & & \\
f ::= x \mid c_s & & \text{(Converter)} \\
c_s \in Const_s & c_c \in Const_c & \text{(Constants)}
\end{array}
$$

Fig. 1. Eliom$_\varepsilon$'s grammar

A *client fragment* $\{\{\ e_c\ \}\}$ stands for an expression computed by the client but that can be referred from the server. An *injection* $f\%e_s$ is used inside client code to access values defined on the server. This involves a serialization on the server followed by a deserialization on the client, which is explicitly specified by a *converter* f. To simplify the semantics, we syntactically restrict converters to be either a variable x or a server constant c_c. We describe converters more precisely in Section 3.2.

Furthermore, we add a validity constraints to our programs: We only consider programs such that variables used under an injection are declared outside of the local client scope, which can be either a client declaration or a client fragment.

3.2 Type system

The type system of ELIOM$_\varepsilon$ is an extension of the regular ML type system. We follow closely [26]. Again, the language is split into a client and a server part.

$$\sigma_\varsigma ::= \forall\alpha^*.\tau_\varsigma \qquad \text{(TypeSchemes)}$$
$$\tau_s ::= \alpha \mid \tau_\text{s} \to \tau_\text{s} \mid \{\tau_\text{c}\} \mid \tau_s \rightsquigarrow \tau_c \mid \kappa \text{ for } \kappa \in \mathit{ConstType}_s$$
$$\tau_c ::= \alpha \mid \tau_\text{c} \to \tau_\text{c} \mid \kappa \text{ for } \kappa \in \mathit{ConstType}_c \qquad \text{(Types)}$$

$\mathit{ConstType}_\varsigma$ is the set of ground types. Two server-side types are added to core ML types: $\{\tau_\text{c}\}$ is the type of a client fragment whose content is of type τ_c and $\tau_s \rightsquigarrow \tau_c$ is the type of converters from server type τ_s to client type τ_c. This last type is described in more details below. No client-side constructions are added to core ML types: in particular, the type of a client expression can never contain the type of a client fragment $\{\tau_\text{c}\}$.

The typing rules are presented in Figure 2. There are three distinct judgments: $\blacktriangleright$ is the typing judgment for programs, $\triangleright_c$ for client expressions and $\triangleright_s$ for server expressions. $\triangleright_\varsigma$ is used for rules that are valid both on client and server expressions. An environment Γ contains two kinds of bindings: client and server bindings, marked with the subscripts s and c respectively. The instantiation relation is noted by $\sigma \succ \tau$. It means that the type τ is an instance of the type scheme σ. $\mathit{Close}(\tau, \Gamma)$ is the function that closes a type τ over the environment Γ, hence producing a scheme. $\mathit{TypeOf}_\varsigma$ is a map from constants to their types. Most rules are straightforwardly adapted from regular ML rules. The main rules of interest are FRAGMENT and INJECTION: Rule FRAGMENT is for the construction of client fragments. If e_c is of type τ_c in context Γ, then $\{\{\ e_c\ \}\}$ is of type $\{\tau_\text{c}\}$ in the same context. Rule INJECTION is for the communication of server to client. If the server expression e_s is of type τ_s and the converter f is of type $\tau_s \rightsquigarrow \tau_c$, we can use, in a client declaration, the expression $f\%e_s$ with type τ_c. Since no other typing rules involves client fragments, it is impossible to deconstruct them.

Converters To transmit values from the server to the client, we need a serialization format. We assume the existence of a type `serial` in both $\mathit{ConstType}_s$ and $\mathit{ConstType}_c$, which represents the serialization format. The actual format is irrelevant. For instance, one could use JSON or XML.

Converters are special values that describe how to move a value from the server to the client. A converter can be understood as a pair of functions. A converter f of type $\tau_s \rightsquigarrow \tau_c$ is composed of a server-side encoding function of type $\tau_\mathrm{s} \to \texttt{serial}$, and a client-side decoding function of type $\texttt{serial} \to \tau_\mathrm{c}$. We assume the existence of two built-in converters:

- The `serial` converter of type $\texttt{serial} \rightsquigarrow \texttt{serial}$. Both sides are the identity.
- The `fragment` converter of type $\forall\alpha.(\{\alpha\} \rightsquigarrow \alpha)$. Note that this type scheme can only be instantiated with client types.

Type universes It is important to note that there is no identity converter (of type $\forall\alpha.(\alpha \rightsquigarrow \alpha)$). Indeed the client and server type universes are distinct and we cannot translate arbitrary types from one to the other. Some types are only available on one side: database handles, system types, JavaScript API types. Some types, while available on both sides, are simply not transferable. For example, functions cannot be serialized. Finally, some types may share a semantic meaning, but not their actual representation. This is the case where converters are used. For example, integers are often 64-bit on the server and are 32-bit in JavaScript. So, there is an $\texttt{int}_s$ and an $\texttt{int}_c$ type, along with a converter of type $\texttt{int}_s \rightsquigarrow \texttt{int}_c$. Another example is an HTTP endpoint. On the server, it is a URL together with a function called when the endpoint is reached.

Common rules

$$\textsc{Var}\ \frac{(x:\sigma)_\varsigma \in \Gamma \qquad \sigma \succ \tau}{\Gamma \rhd_\varsigma x : \tau} \qquad \textsc{Lam}\ \frac{\Gamma, (x:\tau_1)_\varsigma \rhd_\varsigma e : \tau_2}{\Gamma \rhd_\varsigma \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{Const}\ \frac{\mathit{TypeOf}_\varsigma(c) \succ \tau}{\Gamma \rhd_\varsigma c : \tau}$$

$$\textsc{App}\ \frac{\Gamma \rhd_\varsigma e_1 : \tau_1 \to \tau_2 \qquad \Gamma \rhd_\varsigma e_2 : \tau_1}{\Gamma \rhd_\varsigma (e_1\ e_2) : \tau_2} \qquad \textsc{Let}\ \frac{\Gamma \rhd_\varsigma e_1 : \tau_1 \qquad \Gamma, (x : \mathit{Close}(\tau_1, \Gamma))_\varsigma \rhd_\varsigma e_2 : \tau_2}{\Gamma \rhd_\varsigma \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\textsc{Y}\ \frac{}{\Gamma \rhd_\varsigma \texttt{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2}$$

Server rules

$$\textsc{Fragment}\ \frac{\Gamma \rhd_c e_c : \tau_\mathrm{c}}{\Gamma \rhd_s \{\{\ e_c\ \}\} : \{\tau_\mathrm{c}\}}$$

Client rules

$$\textsc{Injection}\ \frac{\Gamma \rhd_s f : \tau_\mathrm{s} \rightsquigarrow \tau_\mathrm{c} \qquad \Gamma \rhd_s e_s : \tau_\mathrm{s}}{\Gamma \rhd_c f\%e_s : \tau_\mathrm{c}}$$

Eliom$_\varepsilon$'s rules

$$\textsc{Prog}\ \frac{\Gamma \rhd_\varsigma e : \tau_1 \quad \Gamma, (x : \mathit{Close}(\tau_1, \Gamma))_\varsigma \blacktriangleright p : \tau_2}{\Gamma \blacktriangleright \texttt{let}_\varsigma\ x = e\ \texttt{in}\ p : \tau_2} \qquad \textsc{Return}\ \frac{\Gamma \rhd_c e_c : \tau_\mathrm{c}}{\Gamma \blacktriangleright e_c : \tau_\mathrm{c}}$$

$$\mathit{Close}(\tau, \Gamma) = \forall\alpha_0 \ldots \alpha_n.\tau \text{ where } \{\alpha_0, \ldots, \alpha_n\} = \mathit{FreeTypeVar}(\tau) \backslash \mathit{FreeTypeVar}(\Gamma)$$

Fig. 2. Typing rules for Eliom$_\varepsilon$

On the client, it is only the URL of the specified endpoint. These two types are distinct but share the same semantic meaning, and a converter relates them.

Implementation of converters Specifying which converter to use for which injection is quite tedious in practice. The current implementation of Eliom uses runtime information to discover which converter to apply. A better implementation would use ad-hoc polymorphism, such as modular implicits [25] or type classes, to define converters.

3.3 The semantics

We now define an operational semantics for Eliom$_\varepsilon$. The goal of this semantics is to provide a good model of how programs behave. It does not model finer details of the execution like network communication. However, the order of execution is the one a programmer using Eliom should expect. Before defining the semantics, let us provide preliminary definitions. Values are defined in Figure 3. For the evaluation of constants, we assume the existence of two *partial* functions, δ_c and δ_s that interpret the application of a constant to a closed value in order to yield another closed value: $\delta_\varsigma(c_\varsigma, v_\varsigma) = v'_\varsigma$

A queue ξ accumulates the expressions that will have to be evaluated client-side. It contains bindings $[\mathbf{r} \mapsto e_c]$, where $\mathbf{r}$ is a variable. We adopt the convention that bold letters, like $\mathbf{r}$, denote a variable bound to a client expression. The queue is a first-in, first-out data structure. We note $+\!\!+$ the concatenation on queues. Substitute of a variable by a value in an expression or a program is noted $e[v/x]$.

Eliom$_\varepsilon$ is eager and call by value. Evaluation contexts are shown in Figure 3. Injections are rewritten inside client expressions. The location where this can take place is specified by context $C[e_c]$ below. We write $e_c^\star$ for a client expression containing no injection. Thus, we are forcing a left to right evaluation inside client fragments. The evaluation context for expressions $E_\varsigma[e_\varsigma]$ is standard, except that, we also evaluate server expressions in injections $f\%e_s$ inside client code. Program contexts $F[e_\varsigma]$ specifies that the evaluation can take place either in a server declaration or in server code deep inside injections.

The semantics is shown in Figure 4. We define three single-step reduction relations: two relations $\rightarrow$ on expressions indexed by the expression location ς, and the relation $\hookrightarrow$ on programs (or, more precisely on pairs of a program and an environment of execution ξ). We write $\rightarrow^*$ and $\hookrightarrow^*$ for the transitive closures of these relations.

Server declarations are executed immediately (rules Let$_s$ and Context). However client declarations are not. Instead the corresponding expressions are stored in the queue ξ in the order of the program (rule Let$_c$). When encountering a client fragment $\{\{\ e_c\ \}\}$, the expression e_c is not executed at once. Instead, $\{\{\ e_c\ \}\}$ is replaced by a fresh variable $\mathbf{r}$ and e_c is stored in ξ (rule ClientFragment$_s$). When a converter is called inside client code, the encoding part of the converter is executed immediately, while the decoded part is

$$v_\varsigma ::= c_\varsigma \mid x \mid \texttt{Y} \mid \lambda x.e_\varsigma \qquad \text{(Values)}$$

$$F ::= \texttt{let}_s\ x = E_s\ \texttt{in}\ p \mid \texttt{let}_c\ x = C[f\%E_s]\ \texttt{in}\ p \qquad \text{(Program contexts)}$$
$$E_\varsigma ::= [\,] \mid (E_\varsigma\ e_\varsigma) \mid (v_\varsigma\ E_\varsigma) \mid \texttt{let}\ x = E_\varsigma\ \texttt{in}\ e_\varsigma \mid \{\!\{\ C[f\%E_\varsigma]\ \}\!\} \qquad \text{(Expression contexts)}$$
$$C ::= [\,] \mid (C\ e_c) \mid (e_c^\star\ C) \mid \lambda x.C \mid \texttt{let}\ x = C\ \texttt{in}\ e_c \mid \texttt{let}\ x = e_c^\star\ \texttt{in}\ C \qquad \text{(Client contexts)}$$

Fig. 3. ELIOM$_\varepsilon$'s values and evaluation contexts

Common semantics

$$\textsc{App}\ \frac{}{(\lambda x.e_\varsigma)\ v_\varsigma \mid \xi \to e_\varsigma[v_\varsigma/x] \mid \xi} \qquad \textsc{Y}\ \frac{}{\texttt{Y}\ v_\varsigma \mid \xi \to v_\varsigma\ (\lambda x.(\texttt{Y}\ v_\varsigma)\ x) \mid \xi} \qquad \textsc{Let}\ \frac{}{\texttt{let}\ x = v_\varsigma\ \texttt{in}\ e_\varsigma \mid \xi \to e_\varsigma[v_\varsigma/x] \mid \xi}$$

$$\textsc{Delta}\ \frac{\delta_\varsigma(c_\varsigma, v_\varsigma) \text{ is defined}}{(c_\varsigma\ v_\varsigma) \mid \xi \to \delta_\varsigma(c_\varsigma, v_\varsigma) \mid \xi}$$

Server semantics

$$\textsc{Converter}_s\ \frac{f \notin \{\texttt{serial}, \texttt{fragment}\}}{\{\!\{\ C[f\%e_s]\ \}\!\} \mid \xi \underset{s}{\longrightarrow} \{\!\{\ C[\texttt{fragment}\%(\texttt{decode}\ f)\ \texttt{serial}\%((\texttt{encode}\ f)\ e_s)]\ \}\!\} \mid \xi}$$

$$\textsc{Serial}_s\ \frac{f \in \{\texttt{serial}, \texttt{fragment}\}}{\{\!\{\ C[f\%v]\ \}\!\} \mid \xi \underset{s}{\longrightarrow} \{\!\{\ C[v]\ \}\!\} \mid \xi} \qquad \textsc{ClientFragment}_s\ \frac{\mathbf{r} \text{ fresh}}{\{\!\{\ e_c^\star\ \}\!\} \mid \xi \underset{s}{\longrightarrow} \mathbf{r} \mid \xi +\!\!+ [\mathbf{r} \mapsto e_c^\star]}$$

ELIOM$_\varepsilon$'s semantics

$$\textsc{Context}_\varsigma\ \frac{e_\varsigma \mid \xi \underset{\varsigma}{\longrightarrow} e'_\varsigma \mid \xi'}{F[e_\varsigma] \mid \xi \hookrightarrow F[e'_\varsigma] \mid \xi'} \qquad \textsc{Let}_s\ \frac{}{\texttt{let}_s\ x = v\ \texttt{in}\ p \mid \xi \hookrightarrow p[v/x] \mid \xi} \qquad \textsc{Final}_c\ \frac{e \mid \emptyset \underset{c}{\longrightarrow} e' \mid \emptyset}{E_c[e] \mid \emptyset \hookrightarrow E_c[e'] \mid \emptyset}$$

$$\textsc{Converter}_c\ \frac{f \notin \{\texttt{serial}, \texttt{fragment}\}}{\texttt{let}_c\ x = C[f\%e_s]\ \texttt{in}\ p \mid \xi \hookrightarrow \texttt{let}_c\ x = C[\texttt{fragment}\%(\texttt{decode}\ f)\ \texttt{serial}\%((\texttt{encode}\ f)\ e_s)]\ \texttt{in}\ p \mid \xi}$$

$$\textsc{Serial}_c\ \frac{f \in \{\texttt{serial}, \texttt{fragment}\}}{\texttt{let}_c\ x = C[f\%v]\ \texttt{in}\ p \mid \xi \hookrightarrow \texttt{let}_c\ x = C[v]\ \texttt{in}\ p \mid \xi} \qquad \textsc{Let}_c\ \frac{}{\texttt{let}_c\ x = e_c^\star\ \texttt{in}\ p \mid \xi \hookrightarrow p \mid \xi +\!\!+ [x \mapsto e_c^\star]}$$

$$\textsc{Exec}\ \frac{e_c \mid \emptyset \underset{c}{\longrightarrow} e'_c \mid \emptyset}{e \mid [\mathbf{r} \mapsto e_c] +\!\!+ \xi \hookrightarrow e \mid [\mathbf{r} \mapsto e'_c] +\!\!+ \xi} \qquad \textsc{ExecVal}\ \frac{}{e \mid [\mathbf{r} \mapsto v_c] +\!\!+ \xi \hookrightarrow e[v_c/\mathbf{r}] \mid \xi[v_c/\mathbf{r}]}$$

Fig. 4. ELIOM$_\varepsilon$'s operational semantics

transmitted to the client (rules CONVERTER$_s$ and CONVERTER$_c$, followed by CONTEXT$_\varsigma$). The primitive `encode` returns the server side encoding function of a converter; the primitive `decode` returns a reference to a client fragment implementing the client side decoding function of a converter. The `serial` and `fragment` converters are basically the identity, so they are erased once the value to be transferred has been computed (rules SERIAL$_s$ and SERIAL$_c$).

Once all the server declarations have been executed, the expressions in ξ are executed in the same order they were encountered prior in the evaluation (rules EXEC, CONTEXT and BIND). This means that the execution of an ELIOM$_\varepsilon$ program can be split into two phases: server-side execution, then client-side execution. Even though server and client declarations are interleaved in the program, their executions are not. During the first half of the execution, ξ grows as fragments and client code are stored. During the second half of the execution, ξ shrink until it is empty.

This semantics is not equivalent to immediately executing every piece of client code when encountered: the order of execution would be different. The separation of code execution in two stages, the server stage first and the client stage later, allows to properly model common web pattern and to minimize client-server communications. Since execution of stages are clearly separated, only one communication is needed, between the two stage execution. We will see this in more details in the next two sections.

4 Compilation to client and server languages

In a more realistic computation model, different programs are executed on the server and on the client. We thus present a compilation process that separates the server and client parts of ELIOM programs, resulting in purely server-side and client-side programs. We express the output of the compiler in an ML-like language with some specific primitives for both sides. The further compilation of these ML-like languages to machine code for the server, and to JAVASCRIPT for the client [24] is out of the scope of this paper.

4.1 The languages

We define ML$_\text{S}$ and ML$_\text{C}$, two ML languages extended with specific primitives for client-server communication.

$$
\begin{array}{lll}
p ::= & \texttt{let}\ x = e\ \texttt{in}\ p \mid \texttt{bind}\ x = e\ \texttt{in}\ p \mid e & \text{(Programs)} \\
e ::= & v \mid (e\ e) \mid \texttt{let}\ x = e\ \texttt{in}\ e & \text{(Expressions)} \\
v ::= & c \mid x \mid \texttt{Y} \mid \lambda x.e & \text{(Values)} \\
& c \in \mathit{Const} & \text{(Constants)}
\end{array}
$$

Again, the language is parametrized by a set of constants. We assume a constant () of type `unit`. As previously, we write **r** for a variable referring to a client expression, when we want to emphasize this fact for clarity. The language

also contains a `bind` construction. Like a `let` binding, it binds the value of an expression e to a variable x in a program p. However, the variable x is not lexically scoped: you should see it as a global name, that can be shared between the client and the server code.

Primitives A number of primitives are used to pass information from the server to the client. We use globally scoped variables for communication. In an actual implementation, unique identifiers would be used. We use various meta-variables to make the purpose of these global variables clearer. $\mathbf{x}$ is a variable that references an injection, $\mathbf{f}$ references a closure. The server language $\mathrm{ML_S}$ provides these primitives:

- "`injection` $\mathbf{x}$ e" registers that $\mathbf{x}$ corresponds to the injection e.
- "`fragment` $\mathbf{f}$ $\vec{e}$" registers a client fragment to be executed on the client; the code is expected to be bound to $\mathbf{f}$ on the clients and the injections values are given by the vector of expressions $\overline{e}$.
- "`end` ()" signals the end of a server declaration, and hence that there will be no more client fragments from this declaration to execute.

The primitive "`exec` ()" of the client language $\mathrm{ML_C}$ executes the client fragments encountered during the evaluation of the last server declaration.

The primitive `end` () and `exec` () are used to correctly interleave the evaluation of client fragments (coming from server declarations) and client declarations, since server declarations are not present on the client.

4.2 The semantics

The semantics of ML_ε uses similar tools as the semantics of $\textsc{Eliom}_\varepsilon$. The rules for $\mathrm{ML_S}$ and $\mathrm{ML_C}$ are presented in Figure 6. The rules that are common with $\textsc{Eliom}_\varepsilon$ are omitted. A FIFO queue ξ_c records the client fragments to be executed: it contains bindings $[\mathbf{r} \mapsto e]$ as well as a specific token `end` that signals the end of a server-side declaration.

Injections are recorded server-side in an environment γ_{inj} which contains a mapping from the injection reference $\mathbf{x}$ to either a reference $\mathbf{r}$ to the corresponding client fragment or a value of type `serial`. Evaluation contexts are shown in Figure 5.

The semantics for $\mathrm{ML_S}$ is given in Figure 6. It possesses two specific rules, $\textsc{Injection}_s$ and $\textsc{ClientFragment}_s$ which queue injections and client fragments inside respectively γ_{inj} and ξ_c. The rule $\textsc{ClientFragment}_s$ generates a fresh reference $\mathbf{r}$ for each client fragment that will eventually be bound client-side to a value by the $\textsc{ExecVal}_c$ rule.

At the end of the execution of the server program, ξ_c only contains bindings of the shape $[\mathbf{r} \mapsto (\mathbf{f}\ \overline{v})]$. We then transmit the client program and the content of γ_{inj} and ξ_c to the client. Before client execution, we substitute once and for all the injections by their values provided by γ_{inj}: $p'_c = p_c[\gamma_{\mathrm{inj}}]$. We can then execute the client program p'_c.

$$E ::= [\,] \mid (E\ e) \mid (v\ E) \mid \texttt{let}\ x = E\ \texttt{in}\ e \qquad \text{(Evaluation contexts)}$$

Fig. 5. ML_ε's evaluation contexts

Server semantics

$$\frac{\mathbf{r}\text{ fresh} \qquad \xi_c' = \xi_c +\!\!+ [\mathbf{r} \mapsto (\mathbf{f}\ \overline{v})]}{\texttt{fragment}\ \mathbf{f}\ \overline{v} \mid \xi_c, \gamma_{\text{inj}} \underset{s}{\longrightarrow} \mathbf{r} \mid \xi_c', \gamma_{\text{inj}}} \text{ClientFragment}_s$$

$$\frac{\gamma_{\text{inj}}' = \gamma_{\text{inj}} \cup [\mathbf{x} \mapsto v]}{\texttt{injection}\ \mathbf{x}\ v; p \mid \xi_c, \gamma_{\text{inj}} \underset{s}{\hookrightarrow} p \mid \xi_c, \gamma_{\text{inj}}'} \text{Injection}_s$$

$$\frac{\xi_c' = \xi_c +\!\!+ \texttt{end}}{\texttt{end}\ (); p \mid \xi_c, \gamma_{\text{inj}} \underset{s}{\hookrightarrow} p \mid \xi_c', \gamma_{\text{inj}}} \text{End}$$

Client semantics

$$\frac{e \to e'}{\texttt{exec}\ (); p \mid [\mathbf{r} \mapsto e] +\!\!+ \xi \underset{c}{\hookrightarrow} \texttt{exec}\ (); p \mid [\mathbf{r} \mapsto e'] +\!\!+ \xi} \text{Exec}_c$$

$$\frac{}{\texttt{exec}\ (); p \mid \texttt{end} +\!\!+ \xi_c \underset{c}{\hookrightarrow} p \mid \xi_c} \text{ExecEnd}_c \qquad \frac{}{\texttt{exec}\ (); p \mid [\mathbf{r} \mapsto v_c] +\!\!+ \xi \underset{c}{\hookrightarrow} p[v_c/\mathbf{r}] \mid \xi[v_c/\mathbf{r}]} \text{ExecVal}_c$$

$$\frac{}{\texttt{bind}\ \mathbf{f} = v\ \texttt{in}\ p \mid \xi_c \underset{c}{\hookrightarrow} p[v/\mathbf{f}] \mid \xi_c[v/\mathbf{f}]} \text{Bind}$$

Fig. 6. Semantics for ML_s and ML_c

The execution of client fragments is segmented by server-side declaration, materialized client-side by a call to `exec`. Each client fragment coming from the evaluation of the related server declaration is executed in turn through the rules Exec_c, ExecVal_c and Bind. Once no more client fragments coming from this declaration is found in ξ_c, rule ExecEnd_c is applied.

4.3 From Eliom_ε to ML_ε

Before introducing the exact semantics of these primitives, we specify how Eliom_ε is translated using these primitives, which will make their behavior clearer. A key point is that we adopt a distinct compilation strategy for client declarations and for client fragments. Indeed, client declarations are much simpler than client fragments, as they are executed only once and immediately. Their code can be used directly in the client, instead of relying on a sophisticated mechanism.

The rewriting function ρ from Eliom_ε to ML_ε can be split into two functions ρ_s and ρ_c, respectively for server and client code. For each case, we first decompose injections $f\%e_s$ into an equivalent expression:

$$\texttt{fragment}\%(\texttt{decode}\ f)\ \texttt{serial}\%(\texttt{encode}\ f\ e_s)$$

Translating client declarations is done by taking the following steps:

– For each injection $\texttt{fragment}\%e_s$ or $\texttt{serial}\%e_s$, we generate a fresh name $\mathbf{x}$.
– In ML_S, the primitive "`injection` $\mathbf{x}$ e_s" is called for each injection; it signals that the value of x should be transmitted to the client.
– In ML_C, we replace each injection $\texttt{fragment}\%e_s$ or $\texttt{serial}\%e_s$ by $\mathbf{x}$.

An example is presented Figure 7. On the server, the return value of the program is always () since the server program never returns anything. The client program returns the same value as the ELIOM_ε one. The reference to the client fragment implementing the decoder is associated to variable **conv_int**, to be transmitted and used by the client to decode the value.

ELIOM_ε	ML_S	ML_C
$\texttt{let}_s\ x = 1\ \texttt{in}$	`let` $x = 1$ `in`	`let` $y =$
$\texttt{let}_c\ y = \texttt{int}\%x + 1\ \texttt{in}$	`injection` **conv_int** (`decode int`);	(**conv_int** $\mathbf{x}$) $+ 1$
y	`injection` $\mathbf{x}$ ((`encode int`) x)	`in` y

Fig. 7. Example: Compilation of injections

Translating server declarations containing client fragments is a bit more involved. We need to take care of executing client fragments on the client.

– For each client fragment {{ e }} containing the injections $f_i\%e_i$, we create a fresh reference $\mathbf{f}$.
  • In ML_C, we bind $\mathbf{f}$ to "$\lambda x_0, x_1, \ldots.(e')$" where e' is the expression where each injection $\texttt{fragment}\%e_i$ or $\texttt{serial}\%e_i$ has been replaced by x_i.
  • In ML_S, we replace the original client fragment by "`fragment` $\mathbf{f}$ $[e_0, \ldots]$" which encode the injected values and registers that the closure $\mathbf{f}$ should be executed on the client.
– In ML_S, we call "`end` ()" which signals the end of a declaration by sending the `end` token on the queue.
– In ML_C, we call "`exec` ()" which executes all the client fragments, until the `end` token is reached.

An example is presented in Figure 8. You can see that the computation is prepared on the client (by binding a closure), scheduled by the server (by the `fragment` primitive) and then executed on the client (thanks to primitive `exec`). Also note that client fragments without any injection are bound to a closure with a unit argument, in order to preserve side effect order.

A more detailed presentation of the translation rules are given in Appendix A.

ELIOM_ε	ML_S	ML_C
$\texttt{let}_s\ x = \{\{\ 1\ \}\}\ \texttt{in}$	`let` $x =$ `fragment` $\mathbf{f_0}$ [ ] `in`	**bind** $\mathbf{f_0} = \lambda().1$ **in**
	`end` ();	`exec` ();
$\texttt{let}_s\ y = \{\{\ \texttt{fragment}\%x + 1\ \}\}\ \texttt{in}$	`let` $y =$ `fragment` $\mathbf{f_1}$ $[x]$ `in`	**bind** $\mathbf{f_1} = \lambda x.(x + 1)$ **in**
	`end` ();	`exec` ();
...	...	...

Fig. 8. Example: Compilation of client fragments

5 Relating ELIOM$_\varepsilon$ and ML$_\varepsilon$

We need to guarantee that the translation from ELIOM$_\varepsilon$ to the two languages two languages, ML$_\text{S}$ and ML$_\text{C}$ is faithful.

Since ELIOM$_\varepsilon$ is parametrized by its constants, the functions δ, *Const* and *TypeOf* must satisfy a typability condition for the language to be sound.

Hypothesis 1 (δ-typability).
For ς *in* $\{c, s\}$, *if* $\text{TypeOf}_\varsigma(c) \succ \tau' \to \tau$ *and* $\triangleright_\varsigma v : \tau'$
then $\delta_\varsigma(c, v)$ *is defined and* $\triangleright_\varsigma \delta_\varsigma(c, v) : \tau$

We extend the typing relation to account for the execution queue ξ. The judgment $\xi \vDash p : \tau$ states that, given the execution queue ξ, the program p has type τ. We also introduce an judgment $\Gamma \blacktriangleright \xi : \Gamma'$ to type execution queues, where the environment Γ' extends Γ with the types of the bindings in ξ. We introduce the following typing rules.

$$\textsc{Queue}\ \frac{\emptyset \blacktriangleright \xi : \Gamma \qquad \Gamma \blacktriangleright p : \tau}{\xi \vDash p : \tau} \qquad\qquad \textsc{Empty}\ \frac{}{\Gamma \blacktriangleright \emptyset : \Gamma}$$

$$\textsc{Append}\ \frac{\Gamma \triangleright_c e : \tau \qquad \Gamma, (\mathbf{r} : \{\tau\})_s, (\mathbf{r} : \tau)_c \blacktriangleright \xi : \Gamma'}{\Gamma \blacktriangleright [\mathbf{r} \mapsto e] +\!\!+ \xi : \Gamma'}$$

The rule QUEUE tells us that if we can type a queue, producing an environment Γ, and we can type a program p in this environment, then we can type the pair of the queue and the program. This allows us to type a program during its evaluation, and in particular when the queue is no longer empty since the rule CLIENTVALUE has been applied.

Assuming the δ-typability hypothesis, we can now give the following two theorem. This guarantees that the semantics of ELIOM$_\varepsilon$ can be used to reason about side effects and evaluation behaviors in compiled ELIOM$_\varepsilon$ programs.

Theorem 1 (Subject Reduction).
If $\xi_1 \vDash p_1 : \tau$ *and* $p_1 \mid \xi_1 \hookrightarrow p_2 \mid \xi_2$ *then* $\xi_2 \vDash p_2 : \tau$.

Theorem 2 (Simulation). *Let* p *be an* ELIOM$_\varepsilon$ *program with an execution* $p \mid \emptyset \hookrightarrow^* v \mid \emptyset$ *For an execution of* p *that terminates, we can exhibit a chained execution of* $\rho_s(p)$ *and* $\rho_c(p)$ *such that evaluation is synchronized with* p.

6 Related work

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. JavaScript can be used on the server, with Node.js; it can be used as a compilation target: for instance, 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 simple 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.

Tierless languages and libraries Several other languages share with Eliom the characteristic of mixing client and server code in an almost transparent way. We will first give a high-level comparison of the various trade-offs involved.

In Eliom, code location is indicated by manual annotations. Several other approaches infer code location using known elements (database access is on the server, dynamic DOM interaction is done on the client, etc) and control flow analysis [18,17,9] . This approach presents various difficulties and drawbacks: It is extremely difficult to integrate to an existing language; it is difficult to achieve with an effectful language; the slicing cannot be as precise as explicit annotations. For example it will not work if the program builds datastructures that mix client fragments and other data, as shown in Section 2.4. 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 two type universes for client and server types (see Section 3.2). This allows the type system to check which functions and types are usable on which side. Most other systems do not track such properties at the type level.

Eliom uses asymmetric communication between client and server (see Section 3.3). Most other languages provide only two-way communications. The actual implementation of Eliom also provides two-way communications as a library, allowing the user to use them when appropriate.

We now provide an in-depth comparison with the most relevant approaches.

Ur/Web [8,7] is a new statically typed language special purposed for Web programming. While similar in scope to Eliom, it presents a very different approach: Ur/Web uses whole-program compilation and a global automatic slicing to separate client and server code. This makes some examples hard to express, such as the one in Section 2.4. 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 and contrary to Eliom, several primitives such as RPC are hardcoded in the language.

Hop [21,6] is a dialect of Scheme for programming Web applications. Like Eliom it uses explicit location annotations and provide facilities to write complex client-server applications. However, as a Scheme-based language, it does not provide static typing. In particular, contrary to Eliom, Hop does not enforce statically the separation of client and server universes (such as using database code inside the client).

Links [10] 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 mechanisms to separate client and server code, so they are shared by default, but 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 [11] 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 in the server and the client, which is not generally true.

Meteor.js [15] is a framework to write both the client and the serve side of an application in JavaScript. It has no built-in mechanism for sections and fragments but relies on `if` statements on the `Meteor.isClient` and `Meteor.isServer` constants. 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. Compared to Eliom, this solution only provides coarse grain composition.

MetaOCaml [14] is an extension of OCaml for meta programming, it introduces a quotation annotation for staged expressions for which execution will be delayed. While having a different goal, stage quotations are very similar to Eliom's client fragments. The main difference is the choice of universes: Eliom possesses two universes, client and server, that are distinct. MetaOCaml possesses a series of universes for each stage, included in one another.

7 Conclusion

We have presented a formalization of the core language Eliom$_\varepsilon$, a client-server Web application programming language. First, we have given a formal semantics and a type system for a language that contains the key features of Eliom. This semantics is intuitive and easy to understand. It corresponds to what a programmer needs to know. Then, we have defined a lower level semantics, corresponding to how Eliom is compiled. We then showed how this compilation is done and that it preserves the semantics.

ELIOM$_\varepsilon$ is used as a core language for the larger ELIOM framework and the OCSIGEN ecosystem. ELIOM$_\varepsilon$ 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, to all kinds of widgets and libraries used 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 benefit from an already large community of users. Web programming is never about the Web per se, but almost always related to other fields for which dedicated libraries are necessary.

Explicit annotations indicate at which location the 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 by mistake unwanted communication.

ELIOM makes strong use of static typing to guarantee many properties of the program at compile time. Developing both the client and server parts as a single program allows to guarantee the consistency between the two parts, to check all communications: injections, server push, 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

1. Balat, V.: Client-server Web applications widgets. In: WWW'13 dev track (2013)
2. Balat, V.: Rethinking traditional web interaction: Theory and implementation. International Journal on Advances in Internet Technology (2014)
3. Balat, V., Chambart, P., Henry, G.: Client-server Web applications with Ocsigen. In: WWW'12 dev track. p. 59. Lyon, France (Apr 2012)
4. Balat, V., Vouillon, J., Yakobowski, B.: Experience report: Ocsigen, a Web programming framework. In: ICFP. pp. 311–316. ACM (2009)
5. BeSport. http://www.besport.com/
6. Boudol, G., Luo, Z., Rezk, T., Serrano, M.: Reasoning about Web applications: An operational semantics for HOP. Trans. Program. Lang. Syst. 34(2), 10 (2012)
7. Chlipala, A.: An optimizing compiler for a purely functional Web-application language. In: ICFP (2015)
8. Chlipala, A.: Ur/Web: A simple model for programming the Web. In: POPL (2015)

9. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web applications via automatic partitioning. In: SOSP'07 (2007)
10. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: FMCO. pp. 266–296 (2006)
11. Ekblad, A., Claessen, K.: A seamless, client-centric programming model for type safe web applications. In: SIGPLAN Symposium on Haskell. Haskell '14 (2014)
12. Eliom web site. http://ocsigen.org/
13. New York University Gencore. http://gencore.bio.nyu.edu/
14. Kiselyov, O.: The design and implementation of BER MetaOCaml - System description. In: FLOPS (2014)
15. Meteor.js. http://meteor.com
16. Ocsigen Toolkit. http://ocsigen.org/ocsigen-toolkit/
17. Opa web site. http://opalang.org/
18. Philips, L., De Roover, C., Van Cutsem, T., De Meuter, W.: Towards tierless Web development without tierless languages. In: Onward! 2014 (2014)
19. Pumgrana. http://www.pumgrana.com/
20. Scherer, G., Vouillon, J.: Macaque : Interrogation sûre et flexible de base de données depuis OCaml. In: 21ème journées francophones des langages applicatifs (2010)
21. Serrano, M., Queinnec, C.: A multi-tier semantics for Hop. Higher-Order and Symbolic Computation 23(4), 409–431 (2010)
22. Tyxml. http://ocsigen.org/tyxml/
23. Vouillon, J.: Lwt: a cooperative thread library. In: ACM Workshop on ML (2008)
24. Vouillon, J., Balat, V.: From bytecode to JavaScript: the Js_of_ocaml compiler. Software: Practice and Experience 44(8), 951–972 (2014)
25. White, L., Bour, F., Yallop, J.: Modular implicits. ML workshop (2014)
26. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and computation 115(1), 38–94 (1994)

A Translation from Eliom$_\varepsilon$ to ML$_\varepsilon$

We define two rewriting functions, ρ_s and ρ_c, which take as input an Eliom$_\varepsilon$ program and output respectively an ML$_\mathrm{S}$ and an ML$_\mathrm{C}$ program.

We define some preliminaries notations. Brackets [and] are used as meta-syntactic markers for repeated expressions: $[\dots]_i$ is repeated for all i. The bounds are usually omitted. Lists are denoted with an overline: $\overline{x_i}$ is the list $[x_0, \dots, x_{n-1}]$. We allow the use of lists inside substitutions to denote simultaneous independent substitutions. For instance $e[\overline{v_i}/\overline{x_i}]$ is equivalent to $e[v_0/x_0]\dots[v_{n-1}/x_{n-1}]$. We will only consider substitutions where the order is irrelevant.

We also consider two new operations:

- $injections(e_c)$ returns the list of injections in e_c.
- $fragments(e_s)$ returns the list of fragments in e_s.

The order corresponds to the order of execution for our common subset of ML.

As in Section 4.3, we assume that fragments are first rewritten in the following manner:

$$f\%e_s \longrightarrow \texttt{fragment}\%(\texttt{decode}\ f)\ \texttt{serial}\%(\texttt{encode}\ f\ e_s)$$

We also assume that expressions inside injections are hoisted out of the local client scope. For example $\{\{\ f\%(3{+}x)\ \}\}$ is transformed to $\lambda y.\{\{\ f\%y\ \}\}\ (3{+}x)$. More formally, we apply the following transformations:

$$\{\{\ C[f\%e_s]\ \}\} \longrightarrow (\lambda x.C[f\%x])\ e_s$$
$$\texttt{let}_c\ y = C[f\%e_s]\ \texttt{in}\ p \longrightarrow \texttt{let}_s\ x = e_s\ \texttt{in}\ \texttt{let}_c\ y = C[f\%x]\ \texttt{in}\ p$$
$$\text{Where } x \text{ is fresh.}$$

This transformation preserves the semantics, thanks to the validity constraint on programs presented in Section 3.1.

We can now define ρ_s and ρ_c by induction over ELIOM$_\varepsilon$ programs. We refer to Section 4.3 for a textual explanation.

Since we already translated custom converters, all the converters considered are either `fragment` or `serial`. Here is the definition for client sections:

$$\rho_s(\texttt{let}_c\ x = e_c\ \texttt{in}\ p) \equiv \left[\texttt{injection}\ \mathbf{x}_i\ e_i;\right]_i\ \rho_s(p)$$
$$\rho_c(\texttt{let}_c\ x = e_c\ \texttt{in}\ p) \equiv \texttt{let}\ x = e_c[\overline{\mathbf{x}_i}/\overline{f_i\%e_i}]\ \texttt{in}\ \rho_c(p)$$
$$\text{Where } \begin{cases} \overline{f_i\%e_i} = \mathit{injections}(e_c) \\ \forall i, e_i \text{ does not contain fragments.} \\ \overline{\mathbf{x}_i} \text{ is a list of fresh variables.} \end{cases}$$

Here is the definition for server sections. Note the presence of lists of lists, to handle injections inside each fragment.

$$\rho_s(\texttt{let}_s\ x = e_s\ \texttt{in}\ p) \equiv \texttt{let}\ x = e_s[\texttt{fragment}\ \mathbf{f}_i\ \overline{a}_i/\{\{\ e_i\ \}\}]_i\ \texttt{in}\ \texttt{end}();\ \rho_s(p)$$
$$\rho_c(\texttt{let}_s\ x = e_s\ \texttt{in}\ p) \equiv \left[\texttt{bind}\ \mathbf{f}_i = \lambda(\overline{x})_i.(e_i[(\overline{x})_i/(\overline{f\%a})_i])\ \texttt{in}\ \right]_i\texttt{exec}();\ \rho_c(p)$$
$$\text{Where } \begin{cases} \overline{\{\{\ e_i\ \}\}} = \mathit{fragments}(e_s) \\ \overline{\mathbf{f}_i} \text{ is a list of fresh variables.} \\ \forall i, (\overline{f\%a})_i = \mathit{injections}(e_i) \\ \forall i, (\overline{x})_i \text{ is a list of fresh variables;} \end{cases}$$

Finally, the returned expression of an ELIOM$_\varepsilon$ program. The translation is similar to client sections:

$$\rho_s(e_c) \equiv \left[\texttt{injection}\ \mathbf{x}_i\ e_i;\right]_i\ ()$$
$$\rho_c(e_c) \equiv e_c[\overline{\mathbf{x}_i}/\overline{f_i\%e_i}]$$
$$\text{Where } \begin{cases} \overline{f_i\%e_i} = \mathit{injections}(e_c) \\ \forall i, e_i \text{ does not contain fragments.} \\ \overline{\mathbf{x}_i} \text{ is a list of fresh variables.} \end{cases}$$