Tierless Modules

Gabriel Radanne, Jérôme Vouillon

Tierless Modules

GABRIEL RADANNE, Univ Paris Diderot, Sorbonne Paris Cité
JÉRÔME VOUILLON, Univ Paris Diderot, Sorbonne Paris Cité, BeSport, and CNRS

Tierless Web programming languages allow to combine client-side and server-side programming in a single program. This allows to define expressions with both client and server parts, and at the same time provides good static guarantees regarding client-server communication. However, these nice properties come at a cost: most tierless languages offer very poor support for modularity and separate compilation.

To regain this modularity and offer a larger-scale notion of composition, we propose to leverage a well-known tool: ML-style modules. In this article, we show how to extend the OCaml module system with tierless annotations that specify whether some definitions should be on the server, on the client, or both.

In modern ML languages, the module system is a layer separate from the expression language. Our work relies on Eliom for the expression language. Eliom is an ML tierless Web programming language that provides type-safe communication and an efficient execution model. We complement that with a module language that preserves all the desirable properties of Eliom in terms of typing and efficiency, allows separate compilation, integrates well with the vanilla OCaml module language, and supports datatype abstraction.

Additional Key Words and Phrases: Web, client/server, OCaml, ML, Eliom, functional, module

ACM Reference format:
Gabriel Radanne and Jérôme Vouillon. 2016. Tierless Modules. 1, 1, Article 1 (January 2016), 26 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Traditional Web applications are composed of several distinct tiers: Web pages are written in HTML and styled in CSS; these pages are produced by a server which can be written in about any language: PHP, Ruby, C++ …; their dynamic behavior is controlled through client-side languages such as JavaScript. The traditional way to compose these languages is to write separate programs for the client and the server. Then, the programmer is expected to respect a common interface between the two programs. This constraint is usually not checked automatically, and it is the responsibility of the programmer to ensure that the two programs behave in a coherent manner. Of course, such checking is often error-prone. This issue, present in the Web since its inception, has become even more relevant in modern Web applications. Furthermore, the unit of composition here is a whole file (or compilation unit): files contain either client code or server code but can not be composed of both client and server code. Such composition is very coarse-grained and hinders the modularity of Web programming libraries.

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. The so-called *tierless* languages aim to solve such modularity issues by allowing to compose tiers inside expressions, by allowing to

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

freely intersperse the client and server parts of the application in one language with seamless communication. For most of these languages, the program is sliced in two: a part which runs on the server and a part which is compiled to JavaScript and runs on the client. This allows to write libraries and widgets with both client and server behaviors. It also provides static guarantees about client-server separation and a fine-grained notion of composition.

However, programming of large-scale software and libraries still requires a form of larger-scale composition. Indeed, parts of a library could be entirely on the server or on the client. Most tierless languages do not support such modular approach to program architecture and do not handle separate compilation. To solve this problem, we propose to leverage a well-known tool: ML-style modules. In this article, we show how to extend the ML module system with tierless annotations. By doing so, we gain a convenient paradigm for organizing large-scale software and support for separate compilation on top of the gains provided by tierless programming languages. Our module system is built as a complement to Eliom, a tierless web programming language built on top of OCaml, and retains its good properties such as static typing of client-server communications and an efficient excution model.

1.1 Modules

In modern ML languages, the module language is separate from the expression language. While the language of expression allows to program "in the small", the module language allows to program "in the large". In most languages, modules are compilation units: a simple collection of type and value declarations in a file. The SML module language (MacQueen 1984) uses this notion of collection of declarations (called *structure*) and extends it with types (module specifications, or *signatures*), functions (parametrized modules, or *functors*) and function application, forming a small typed functional language. Such a module system can account for separate compilation (Leroy 1994) and provides support for datatype abstraction (Crary 2017; Leroy 1995), which allows to hide the implementation of a given type in order to enforce some invariants. In the history of ML-programming languages, ML-style modules have been informally shown to be very expressive tools to architecture software. Functors, in particular, allow to write generic implementations by abstracting over a complete module. The OCaml language provides such a module system, extended with various other constructs such as package types (Russo 2000) (also known as first-class modules). One distinctive feature is that modules in OCaml are runtime entities. In contrast to systems such as MLton (2014), they are not eliminated at compile time.

1.2 Eliom

Eliom (Radanne et al. 2016a,b) is an extension of OCaml for tierless programming that supports composable and typesafe client-server interactions. It provides fine-grained modularity by allowing to manipulate *on the server*, as first class values, fragments of code which will be executed *on the client*. Eliom is part of the larger Ocsigen (Balat et al. 2009; Eliom 2017) project, which also includes the compiler js_of_ocaml (Vouillon and Balat 2014), 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 (Ocsigen Toolkit 2017); a powerful session mechanism and an advanced *service identification mechanism* (Balat 2014). The Ocsigen project started in 2004, as a research project, with the goal of building a complete industrial-strength framework.

1.3 A module language for tierless programming languages

All of the modules and libraries in Ocsigen, and in particular in the Eliom framework, are implemented on top of a core language described in Radanne et al. (2016a). The design of this core language is guided by four complementary goals: easy composition of client and server code, type-safe communication between client and server, explicit communications that are easy to reason about and efficient execution model. We introduce additional properties that will drive the design of our module language:

Integration with the host language. Eliom is an extension of OCaml. We should be able to leverage both the language and the ecosystem of OCaml. OCaml libraries can be useful on the server, on the client or on both. As such, any OCaml file, even when compiled with the regular OCaml compiler, is a valid Eliom module. Furthermore, we can specify if we want to use a given library on the client, on the server, or everywhere.

Abstraction. Module languages are very powerful abstraction tools. By only exposing part of a module, the programmer can safely hide implementation details and enforce specific properties. Eliom leverages module abstraction to provide encapsulation and separation of concern for widgets and libraries. By combining module abstraction and tierless features, library authors can provide good APIs that do not expose the minute details of client-server communication to the users.

These properties lead us to define a module language, Eliom$_m$, that extends the tierless core language presented in Radanne et al. (2016a). We give a quick presentation of Eliom from a programming point of view in Section 2. The expression language is defined in Section 3, which reformulates the key ideas of Radanne et al. (2016a). The module language is described in Section 4. Finally we present the target languages in Section 5 and a compilation scheme in Section 6.

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. 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.

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 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 ]
```

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 Modules

One can define client and server modules. One can also use regular OCaml modules and functors inside client and server code. Here, we use `Map.Make` on the client to define maps whose keys are JavaScript strings. `Map.Make` is a pre-defined functor in the OCaml standard library that takes a module implementing the `COMPARABLE` signature as argument and returns a module that implements associative maps (also called dictionaries) whose keys are of the type `t` in the provided module. Since associative maps are implemented using balanced binary trees, a comparison function has to be provided by its argument. Note here that a functor from vanilla OCaml is applied to a client module and returns a client module.

```
module%client JStr = struct
  type t = Js.string
  let compare = Js.compare_string
end

module%client MapJStr = Map.Make(JStr)
```

```
module type COMPARABLE = sig
  type t
  val compare : t -> t -> int
end

module Make (Key : COMPARABLE) : sig
  type 'a t
  val add : Key.t -> 'a -> 'a t -> 'a t
  (* ... *)
end
```

It is also possible to define *mixed* modules, that can contain both client and server declarations. The namespaces for client and server declarations are distinct.

```
module%mixed M = struct
  type%client t = int
  type%server t = t fragment
  let%server x : t = [%client 2]
end
```

We can also defined mixed functors that accept mixed modules as arguments, and produce modules that are themselves mixed. As an example, we build on `Map.Make` to produce *mixed* tables, *i.e.,* tables that have a meaning on both sides. The idea is that adding an entry to a server-side table also adds the element to the client-side table. Consequently, the server-side representation of a table needs to include a client-side one. Such mixed tables are useful, for example, to give the client access to a cache of elements that were already produced by the server while serving the request, thus preventing duplicated work. For brevity, we omit operations other than addition, but note that lookup is easy to implement as lookup on the local table.

```
module%mixed MakeShared (Comparable : COMPARABLE) =
struct
  module%client M = Map.Make(Comparable)
  module%server M = Map.Make(Comparable)

  type%client 'a table = 'a M.t
  type%server 'a table = 'a M.t * 'a M.t fragment

  let%client add id v tbl =
    M.add id v tbl
  let%server add id v (tbl_server, tbl_client) =
    [%client M.add ~%id ~%v ~%tbl_client ];
    M.add id v tbl_server

  (* ... *)
end
```

```
module%mixed MakeShared (Comparable : COMPARABLE) :
sig
  type%client 'a table
  type%server 'a table

  val%client add :
    Comparable.t -> 'a -> 'a table -> 'a table
  val%server add :
    Comparable.t -> 'a -> 'a table -> 'a table

  (* ... *)
end
```

Applying `MakeShared` produces a module containing types (like `table`) and values (like `add`) available on the side the functor is called on. However, the functor cannot be implemented in a decomposed way, given that the server implementation relies on the client-side version of the functor argument (`Comparable`).

Contrary to client and server functors, mixed functors are limited: arguments must be mixed modules and injections inside client-side bindings can only reference elements out of the functor. Additionally, it is forbidden to nest mixed structures and functors arbitrarily.

More complex examples of libraries built with Eliom and Ocsigen can be found in Radanne et al. (2016b).

2.5 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 THE EXPRESSION LANGUAGE

We now present the expression language. This subset is based on the Eliom$_\varepsilon$ language presented by Radanne et al. (2016a). We reformulate it here with several extensions. The notations and presentation of the type system are borrowed in part from Wright and Felleisen (1994).

Let us first define some notations and meta-syntactic variables: e are expressions, v are variables, p are paths, x are module variables, τ are type expressions and t are type constructors. v_i, x_i and $\mathrm{t_i}$ are identifiers (for values, modules and types). Identifiers (such as x_i) have a name part (x) and a stamp part (i) that distinguish identifiers with the same name. α-conversion should keep the name intact and change only the stamp. We use subscripts to specify locations. Core locations, defined in Figure 1, can be either **s**erver, **c**lient or **b**ase. The base side represents expressions that are "location-less", that is, which can be used everywhere. We use the meta-variable ℓ for an unspecified core location.

As Eliom$_m$ is an extension of classic ML with modules, we distinguish new elements in the type system in blue. This is purely for ease of reading and is not essential for understanding this article.

3.1 Syntax

The syntax is presented in Figure 1. The Eliom$_m$ expression language is an extension of a simple ML language with let bindings, polymorphism and generalization. Two additional constructs are introduced for client fragments and injections. The language is parametrized by its constants. There are three sets of constants: base, server and client. An expression or type expression can be either client, server or base.

A *client fragment* $\{\{\ e\ \}\}$ can be used on the server to represent an expression that will be computed on the client, but which future value can be manipulated on the server. An *injection* $f\%v$ can be used on the client to access values defined on the server. An injection must make explicit use of a converter f that specifies how to send the value. In particular, this should involve a serialization step, executed on the server, followed by a deserialization step executed

Locations

$$\ell ::= s \mid c \mid b \qquad \text{(Core)}$$
$$m ::= \varsigma \mid \ell \qquad \text{(Mixed)}$$

Path

$$p ::= x_i \mid p.x \mid p_1(p_2)$$

Expressions

$$e ::= c \mid v_i \mid \mathtt{Y} \mid (e\ e) \mid \lambda v.e$$
$$\mid \mathtt{let}\ v = e\ \mathtt{in}\ e \qquad \text{(Let)}$$
$$\mid p.v \qquad \text{(Path)}$$
$$\mid \{\{\ e\ \}\} \qquad \text{(Fragments)}$$
$$\mid f\%v \qquad \text{(Injections)}$$
$$f ::= p.v \mid v_i \mid c \qquad \text{(Converter)}$$
$$c \in \mathit{Const}_\ell \qquad \text{(Constants)}$$

Environments

$$\Gamma ::= \emptyset$$
$$\mid \Gamma; (\mathtt{module}\ x : \mathrm{M})_m \qquad \text{(Module)}$$
$$\mid \Gamma; (v : \sigma)_\ell \qquad \text{(Value)}$$
$$\mid \Gamma; (\mathrm{t} = \tau)_\ell \qquad \text{(Type)}$$
$$\mid \Gamma; (\mathrm{t})_\ell \qquad \text{(Abstract Type)}$$

Type Schemes

$$\sigma ::= \forall \alpha^*_{\ell_i}.\tau$$

Type Expressions

$$\tau ::= \alpha_\ell \mid \tau \to \tau \mid \mathrm{t_i}$$
$$\mid p.t \qquad \text{(Path)}$$
$$\mid \{\tau\} \qquad \text{(Fragments)}$$
$$\mid \tau \rightsquigarrow \tau \qquad \text{(Converters)}$$

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

on the client. For ease of presentation, injections are only done on variables and constants. In the implementation, this restriction is removed by adding a lifting transformation (See Section 7.2). For clarity, we sometimes distinguish injections *per se*, which occur outside of fragments, and escaped values, which occur inside fragments.

The syntax of types is also extended with two constructs. A *fragment type* {e} is the type of a fragment. A *converter type* $\tau_s \rightsquigarrow \tau_c$ is the type of a converter taking a server value of type τ_s and returning a client value of type τ_c. Finally, all type variables α_ℓ are annotated with a core location ℓ.

3.2 Type system

We note $TypeOf_\ell(c)$ the type of a given constant c on the location ℓ. The typing judgment is noted $\Gamma \triangleright_\ell e : \tau$ where e is of type τ in the environment Γ on the location ℓ. The typing rules are defined in Figure 3. We note $\sigma \succ \tau$ when the type τ is an instance of the type scheme σ. We note $Close(\Gamma, \tau)$ the function that closes the type τ over the environment Γ and produces a type scheme.

Typing environments Γ can contain four kind of bindings: variable bindings $(v : \tau)_\ell$, type declarations $(\mathrm{t} = \tau)_\ell$, abstract type declarations $(\mathrm{t})_\ell$ and module bindings $(\texttt{module}\ x : \mathrm{M})_m$. All bindings are annotated with a location ℓ. Names are namespaced by locations. The first three kind of bindings, corresponding to the core language, can only appear on core locations: s, c or b. Modules can also be of mixed location ς. We use m as meta variable for locations that can be either ς or another core location.

Most of the rules are straightforward adaptions of traditional ML rules. The two new rules are Fragment and Injection. Rule Fragment is for the construction of client fragments and can only be applied on the server. If e is of type τ *on the client*, the {{ e }} is of type $\{\tau\}$ *on the server*. Rule Injection is for the communication from the server to the client and can only be applied on the client. If e is of type τ_s on the server and f is of type $\tau_s \rightsquigarrow \tau_c$ on the server, then $f\%e$ is of type τ_c on the client. Since no other typing rules involves client fragments, it is impossible to deconstruct them.

The last difference with usual ML rules are the visibility of variable. As described earlier, bindings in Eliom$_m$ are located. Since access across sides are explicit, we want to prevent the use of client variables on the server, for example. In rule Var, to use on location ℓ the variable v which is bound on location ℓ', we check that location ℓ' encompass location ℓ, noted $\ell' \succ \ell$. This relation is defined in Figure 2. Base elements b are usable everywhere. Mixed elements ς are usable in both client and server. Type variables are also annotated with a location and follow the same rules. Using type variables from the client on the server, for example, is disallowed.

We also define two other judgments. The well-formedness judgment on types is noted $\Gamma \vDash_\ell \tau$ where τ is well formed in the environment Γ on the location ℓ. The well-formedness rules are defined in Figure 4. A well formed type is a type with proper usage of locations: client fragments and converters should be used only on the server and reference client types. The type equivalence judgment on types is noted $\Gamma \triangleright_\ell \tau \approx \tau'$ where τ is equivalent to τ' in environment Γ on the location ℓ. We omit the classical rules of ML, along with rules for reflexivity, commutativity and transitivity.

3.2.1 Converters. To transmit values from the server to the client, we need a serialization format. We assume the existence of a type `serial` in $Const_b$ 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 $\tau_s \to \texttt{serial}$, and a client-side decoding function of type $\texttt{serial} \to \tau_c$. 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 `frag` converter of type $\forall\alpha_c.(\{\alpha_c\} \rightsquigarrow \alpha_c)$.

3.2.2 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 (because they are in *b*ase for examle), are simply not transferable. For example, functions cannot be serialized in general. Another example is file handlers: they are available both on the server and on the client, but moving a file handle from server to client seems adventurous.

Finally, some types may share a semantic meaning, but not their actual representation. This is the case where converters are used. Advanced practical usages of converters and separated type universes to create useful libraries can be found in Radanne et al. (2016b).

4 THE MODULE LANGUAGE

We now present the module language part of Eliom$_m$. The module language is an extension of Leroy (1995). The intent here is to model a simple extension of most of the OCaml module language, our eventual goal being to actually implement this extension.

$$\varsigma \succ s \qquad \varsigma \succ c \qquad b \succ s \qquad b \succ c \qquad b \succ \varsigma \qquad \forall \ell \in \{s, c, \varsigma, b\}\ \ell \succ \ell$$

Fig. 2. "can be used in" relations on locations – $\ell' \succ \ell$

Common rules

$$\textsc{Var}\ \frac{(v : \sigma)_{\ell'} \in \Gamma \qquad \ell' \succ \ell \qquad \sigma \succ \tau}{\Gamma \triangleright_\ell v : \tau} \qquad \textsc{Lam}\ \frac{\Gamma; (v : \tau_1)_\ell \triangleright_\ell e : \tau_2}{\Gamma \triangleright_\ell \lambda v.e : \tau_1 \to \tau_2} \qquad \textsc{Const}\ \frac{\mathit{TypeOf}_\varsigma(c) \succ \tau}{\Gamma \triangleright_\ell c : \tau}$$

$$\textsc{LetIn}\ \frac{\Gamma \triangleright_\ell e_1 : \tau_1 \qquad \Gamma; (v : \mathit{Close}(\tau_1, \Gamma))_\ell \triangleright_\ell e_2 : \tau_2}{\Gamma \triangleright_\ell \texttt{let}\ v = e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \textsc{Equiv}\ \frac{\Gamma \triangleright_\ell e : \tau_1 \qquad \Gamma \triangleright_\ell \tau_1 \approx \tau_2}{\Gamma \triangleright_\ell e : \tau_2} \qquad \textsc{App}\ \frac{\Gamma \triangleright_\ell e_1 : \tau_1 \to \tau_2 \qquad \Gamma \triangleright_\ell e_2 : \tau_1}{\Gamma \triangleright_\ell (e_1\ e_2) : \tau_2}$$

$$\textsc{Y}\ \frac{}{\Gamma \triangleright_\ell \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2}$$

Server rules

$$\textsc{Fragment}\ \frac{\Gamma \triangleright_c e : \tau}{\Gamma \triangleright_s \{\{\ e\ \}\} : \{\tau\}}$$

Client rules

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

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

Fig. 3. Typing rules for the expression language – $\Gamma \triangleright_\ell e : \tau$

$$\textsc{DatatypeVal}\ \frac{(\mathrm{t} = \tau)_{\ell'} \in \Gamma \qquad \ell' \succ \ell}{\Gamma \vDash_\ell \mathrm{t}} \qquad \textsc{VarVal}\ \frac{\ell_i \succ \ell}{\Gamma \vDash_\ell \alpha_{\ell_i}} \qquad \textsc{ArrowVal}\ \frac{\Gamma \vDash_\ell \tau_1 \qquad \Gamma \vDash_\ell \tau_2}{\Gamma \vDash_\ell \tau_1 \to \tau_2}$$

$$\textsc{FragmentVal}\ \frac{\Gamma \vDash_c \tau}{\Gamma \vDash_s \{\tau\}} \qquad \textsc{ConvVal}\ \frac{\Gamma \vDash_s \tau_1 \qquad \Gamma \vDash_c \tau_2}{\Gamma \vDash_s \tau_1 \rightsquigarrow \tau_2}$$

Fig. 4. Validity rules for types – $\Gamma \vDash_\ell \tau$

$$\textsc{FragmentEq}\ \frac{\Gamma \triangleright_c \tau \approx \tau'}{\Gamma \triangleright_s \{\tau\} \approx \{\tau'\}} \qquad \textsc{ConvEq}\ \frac{\Gamma \triangleright_s \tau_s \approx \tau_s' \qquad \Gamma \triangleright_c \tau_c \approx \tau_c'}{\Gamma \triangleright_s \tau_s \rightsquigarrow \tau_c \approx \tau_s' \rightsquigarrow \tau_c'} \qquad \textsc{DatatypeEq}\ \frac{(t : \tau)_{\ell'} \in \Gamma \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell t \approx \tau}$$

The classical rules for ML languages are omitted for brevity.

Fig. 5. Type equivalence – $\Gamma \triangleright_\ell \tau \approx \tau'$

4.1 Syntax

Module Expressions

$$\begin{aligned} m ::= {} & x_i \mid p.x \mid m_1(m_2) \mid (m : M) \\ \mid {} & \texttt{struct}\ s\ \texttt{end} \\ \mid {} & \texttt{functor}(x_i : M)m \\ \mid {} & \texttt{functor}_\varsigma(x_i : M)m \end{aligned}$$

Structure body

$$s ::= \varepsilon \mid d; s$$

Structure components

$$\begin{aligned} d ::= {} & \texttt{let}_\ell\ v_i = e_\ell \\ \mid {} & \texttt{type}_\ell\ t_i = \tau_\ell \\ \mid {} & \texttt{module}_m\ x_i = m \end{aligned}$$

Programs

$$P ::= \texttt{prog}\ s\ \texttt{end}$$

Module types

$$\begin{aligned} M ::= {} & \texttt{sig}\ S\ \texttt{end} \\ \mid {} & \texttt{functor}(x_i : M_1)M_2 \\ \mid {} & \texttt{functor}_\varsigma(x_i : M_1)M_2 \end{aligned}$$

Signature body

$$S ::= \varepsilon \mid D; S$$

Signature components

$$\begin{aligned} D ::= {} & \texttt{val}_\ell\ v_i : \tau_\ell \\ \mid {} & \texttt{type}_\ell\ t_i = \tau_\ell \\ \mid {} & \texttt{type}_\ell\ t_i \\ \mid {} & \texttt{module}_m\ x_i : M \end{aligned}$$

Fig. 6. Eliom$_m$'s grammar for modules

The syntax of the module language is given in Figure 6. It is mostly composed of the usual module constructs: functors, module constraints, functor application and structures. A structure is composed of a list of components. Similarly, module types are composed of functors and signatures which are a list of signature components. Components can be declaration of values, types or modules. A type in a signature can be declared abstract or not. The main change in Eliom$_m$ is that structure and signature components are annotated with locations. Value and type declarations can be annotated with a core location ℓ which is either b, s or c. Module declarations can also have one additional possible location: the mixed location ς. We note m a location that can be any of ς, b, s or c. Only modules on location ς can have subfields on different locations. A program is a list of declarations including a client value declaration *res* which is the result of the program.

4.2 Base location and specialization

In Section 2.4, we presented an example where a base functor `Map.Make`, is applied to a client module to obtain a new client module. As `Map.Make` is a module provided by the standard library of OCaml, it is defined on location b. In particular, its input signature has components on location b, thus it would seem a module whose components are on the client or the server should not be accepted. We would nevertheless like to create maps of elements that are only available on the client. To do so, we introduce a specialization operation, defined in Figure 11, that allows to use a base module in a client or server scope by replacing instances of the base location with the current location.

The situation is quite similar to the application of a function of type $\forall\alpha.\alpha \to \alpha$ to an argument of type int: we need to instantiate the functor before being able to use it. The specialization operation simply rewrites a module signature by substituting all instances of the location b or ς by the specified c or s location. Note that before being specialized, a module should be accessible according to the "can be used" relation defined Figure 2. This means that we never have to specialize a server module on the client (or conversely). Specialization towards location b has no effect since only base modules are accessible on location base. Specialization towards the location ς has no effect either: since all locations are allowed inside the mixed location, no specialization is needed. Mixed functors are handled in a specific way, as we see in Section 4.3.

4.3 Mixed Functors

Mixed functors are functors declared in a mixed scope. We note $\texttt{functor}_{\varsigma}(x_i : M)m$ the mixed functor that takes an argument x_i of type M and return a module m. They can contain both client and server declarations (or mixed submodules). Mixed functors and regular functors have different types that are not compatible. Mixed functors have several restrictions compared to regular functors.

4.3.1 Specialization. Specialization on mixed functors only specialize the return type, not the argument. Let us see on an example why this restriction is needed. In Example 1, the functor `F` takes as argument a module containing a base declaration and use it on both sides. If the type of the functor parameter were specialized, the functor application in Example 1b would be well-typed. However, this makes no sense: `M.y` is supposed to represent a fragment whose content is the client value of b, but this value doesn't exist, since b was declared on the server. There would be no value available to inject in the declaration of `y'`.

```
module%mixed F (A : sig val b : int end) = struct
  let%server x = b
  let%server y = [%client b]
end
```

(a) A mixed functor using a base declaration

```
module%server M = F(struct let%server b = 2 end)
let%client y' = ~%M.y
```

(b) An ill-typed application of F

Example 1. A mixed functor using base declaration polymorphically

4.3.2 Injections. Additionally, the body of mixed functors can only contain injections that refer to outside of the functor. Escaped values, which are injections inside client fragments, are still allowed. The functor presented in Example 2a is not allowed while the one in Example 2b is allowed.

This restriction is due to the delayed nature of Eliom and the fact that one cannot determine statically, in the general case, which structure a functor will be applied to. One notable property of injections inside client sections (as opposed to escaped values inside client fragments) is that they are independent of the control flow. Indeed, if an injection is

```
1
2 module%mixed F (A:sig val%server x : int end) =
3 struct
4   let%client y = ~%A.x + 2
5 end
```

(a) An ill-typed mixed functor using an injection

```
1 let%server x = 3
2 module%mixed F (A:sig val%server y : int end) =
3 struct
4   let%client z = ~%x
5   let%server z' = [%client ~%A.y + 1]
6 end
```

(b) A well-typed mixed functor using an injection

Example 2. Mixed functor and injections

syntactically present in an Eliom program, there is only one such injection and its use sites are statically known. We can just give a unique identifier to each injection, and use that unique name for lookup on the client. This property comes from the fact that injected server identifiers can not be bound in a client section. Unfortunately, this property does not hold in the presence of mixed functor when we assume the language can apply functor at arbitrary positions, which is the case in OCaml. We show an example of this in Example 3. The functor F takes a structure containing a server declaration x holding an integer and returns a structure containing the same integer, injected in the client. In Example 3b, the functor is used on A or B conditionally. The issue is that the client integer depends both on the server integer and on the local *client* control flow. Lifting the functor application at toplevel would not preserve the semantics of the language, due to side effects. Thus, we avoid this kind of situation by forbidding injections that access dynamic names inside mixed functors.

```
1 module%mixed F (A : sig val%server x : int end) =
2 struct
3   let%client x' = ~%A.x
4 end
```

(a) An ill-typed mixed functor with an injection

```
1 module%mixed A = struct let%server x = 2 end
2 module%mixed B = struct let%server x = 4 end
3 let%client a =
4   if Random.bool ()
5   then let module M = F(A) in M.x'
6   else let module M = F(B) in M.x'
```

(b) A pathological functor application

Example 3. Ill-typed example of injection inside a mixed functor

4.3.3 Functor application. Mixed functors can only be applied to mixed structures. This means that in a functor application `F(M)`, `M` must be a structure defined by a $\texttt{module}_{\varsigma}$ declaration. Note that this breaks the property that the current location of an expression or a module can be determined syntactically: The location inside `F(struct ... end)` can be either mixed or not, depending on `F`. This could be mitigated by using a different syntax for the application of mixed functor. The justification for this restriction is detailed in Section 6.2.

4.4 Type system

We introduce several type judgments for modules:

$\Gamma \blacktriangleright_m m : M$	The module m is of type M in Γ on the location m. Defined in Figure 8.
$\Gamma \blacktriangleright P : \tau$	The program P returns a client value of type τ. Defined in Figure 8.
$\Gamma \blacktriangleright_m M <: M'$	The module type M is a subtype of the module M' in Γ on ℓ. Defined in Figure 12.
$\Gamma \vDash_m M$	The module type M is well-formed in Γ on ℓ. Omitted for brevity.

We add a new relation on locations, defined in Figure 7: $m <: m'$ means that a module defined on location m can contain component on location m'. In particular, the mixed location ς can contain any component, while other location can contain only component declared on the same location. Additionally, two rules are added to the expression language typing rules to type module paths, shown in Figure 9.

$$\varsigma <: s \qquad \varsigma <: c \qquad \varsigma <: b \qquad \forall \ell \in \{s, c, \varsigma, b\}\ \ell <: \ell$$

Fig. 7. "can contain" relation on sides – $m <: m'$

$$\textsc{Var}\ \frac{(x_i : \mathrm{M})_{m'} \in \Gamma \qquad m' \succ m}{\Gamma \blacktriangleright_m x_i : \lfloor M \rfloor_m} \qquad \textsc{ModVar}\ \frac{\Gamma \blacktriangleright_m p : (\mathtt{sig}\ S_1; \mathtt{module}_{m'}\ x_i : M; S_2\ \mathtt{end}) \qquad m' \succ m}{\Gamma \blacktriangleright_m p.x : \lfloor M\,[n_i \mapsto_{m'} p.n \mid n_i \in \mathrm{BV}_{m'}(S_1)] \rfloor_m} \qquad \textsc{Strength}\ \frac{\Gamma \blacktriangleright_m p : M}{\Gamma \blacktriangleright_m p : M/p}$$

$$\frac{\Gamma \blacktriangleright_m m : M' \qquad \Gamma \blacktriangleright_m M' <: M}{\Gamma \blacktriangleright_m m : M} \qquad \frac{\Gamma \vDash_m M \qquad \Gamma \blacktriangleright_m m : M}{\Gamma \blacktriangleright_m (m : M) : M}$$

$$\frac{\Gamma \vDash_\ell M \qquad x_i \notin \mathrm{BV}_\ell(\Gamma) \qquad \Gamma; (x_i : \mathrm{M})_\ell \blacktriangleright_\ell m : M'}{\Gamma \blacktriangleright_\ell \mathtt{functor}(x_i : M)m : \mathtt{functor}(x_i : M)M'} \qquad \frac{\Gamma \blacktriangleright_m m_1 : \mathtt{functor}(x_i : M)M' \qquad \Gamma \blacktriangleright_m m_2 : M}{\Gamma \blacktriangleright_m m_1(m_2) : M'\,[x_i \mapsto_m m_2]}$$

$$\textsc{MixedFunctor}\ \frac{\Gamma \vDash_\varsigma \mathtt{sig}\ S\ \mathtt{end} \qquad x_i \notin \mathrm{BV}_\varsigma(\Gamma) \qquad \forall f_i \% x_i \in \mathrm{INJS}(m), \Gamma \triangleright_c f_i \% x_i : \tau_i \qquad \Gamma; (x_i : \mathtt{sig}\ \mathrm{S}\ \mathtt{end})_\varsigma \blacktriangleright_\varsigma m : M'}{\Gamma \blacktriangleright_\varsigma \mathtt{functor}_\varsigma(x_i : \mathtt{sig}\ S\ \mathtt{end})m : \mathtt{functor}_\varsigma(x_i : \mathtt{sig}\ S\ \mathtt{end})M'}$$

$$\textsc{MixedApplication}\ \frac{\Gamma \blacktriangleright_m m_1 : \mathtt{functor}_\varsigma(x_i : M)M' \qquad \varsigma \succ m \qquad \Gamma \blacktriangleright_\varsigma m_2 : M}{\Gamma \blacktriangleright_m m_1(m_2) : M'\,[x_i \mapsto_m m_2]}$$

$$\frac{\Gamma \triangleright_\ell e : \tau \qquad m <: \ell \qquad v_i \notin \mathrm{BV}_\ell(\Gamma) \qquad \Gamma; (v : \mathit{Close}(\tau, \Gamma))_\ell \blacktriangleright_m s : S}{\Gamma \blacktriangleright_m (\mathtt{let}_\ell\ v_i = e; s) : (\mathtt{val}_\ell\ v_i : \tau; S)}$$

$$\frac{\Gamma \vDash_\ell \tau \qquad m <: \ell \qquad \mathrm{t_i} \notin \mathrm{BV}_\ell(\Gamma) \qquad \Gamma; (\mathrm{t_i} = \tau)_\ell \blacktriangleright_m s : S}{\Gamma \blacktriangleright_m (\mathtt{type}_\ell\ t_i = \tau; s) : (\mathtt{type}_\ell\ t_i = \tau; S)}$$

$$\frac{\Gamma \blacktriangleright_m m : M \qquad m' <: m \qquad x_i \notin \mathrm{BV}_m(\Gamma) \qquad \forall m'' \in \mathrm{locations}(M).\ m'' \succ m \qquad \Gamma; (\mathtt{module}\ x_i : \mathrm{M})_m \blacktriangleright_{m'} s : S}{\Gamma \blacktriangleright_{m'} (\mathtt{module}_m\ x_i = m; s) : (\mathtt{module}_m\ x_i : M; S)}$$

$$\frac{\Gamma \blacktriangleright_m s : S}{\Gamma \blacktriangleright_m \mathtt{struct}\ s\ \mathtt{end} : \mathtt{sig}\ S\ \mathtt{end}} \qquad \frac{}{\Gamma \blacktriangleright_m \varepsilon : \varepsilon} \qquad \frac{\Gamma \blacktriangleright_\varsigma (\mathtt{struct}\ s\ \mathtt{end}) : (\mathtt{sig}\ \mathtt{val}_c\ res : \tau\ \mathtt{end})}{\Gamma \blacktriangleright (\mathtt{prog}\ s\ \mathtt{end}) : \tau}$$

Fig. 8. Module typing rules – $\Gamma \blacktriangleright_m m : M$

$$\frac{\Gamma \blacktriangleright_\ell p : (\mathtt{sig}\ S_1; \mathtt{val}_{\ell'}\ v_i : \tau; S_2\ \mathtt{end}) \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell p.v : \tau\,[\mathrm{n_i} \mapsto_\ell \mathrm{p.n} \mid \mathrm{n_i} \in \mathrm{BV}_\ell(\mathrm{S_1})]} \qquad \frac{\Gamma \blacktriangleright_\ell p : (\mathtt{sig}\ S_1; \mathtt{type}_{\ell'}\ t_i = \tau; S_2\ \mathtt{end}) \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell p.t \approx \tau\,[n_i \mapsto_\ell p.n \mid n_i \in \mathrm{BV}_\ell(S_1)]}$$

Fig. 9. Additional typing rules for the expression language

The rule Strength uses a strengthening operation noted M/p and defined in Figure 10. A justification and explanation for this operation can be found in Leroy (1994). The rules Var and ModVar follow the usual rules of modules (Leroy 1995) with two modifications: We first check that the module we are looking up can indeed be used on the current location. This is done by the side condition $m' \succ m$ where m is the current location and m' is the location where the identifier is defined. This allows, for instance, to use *base* identifiers in a *client* scope. We also specialize the module

$$\varepsilon/p = \varepsilon \qquad (\texttt{module}_m\ m_i = M; S)/p = \texttt{module}_m\ m_i = M/p; S/p$$
$$(\texttt{sig}\ S\ \texttt{end})/p = \texttt{sig}\ S/p\ \texttt{end} \qquad (\texttt{type}_\ell\ t_i = \tau; S)/p = \texttt{type}_\ell\ t_i = p.t; S/p$$
$$(\texttt{functor}(x_i : M)M')/p = \texttt{functor}(x_i : M)(M'/p(x_i)) \qquad (\texttt{type}_\ell\ t_i; S)/p = \texttt{type}_\ell\ t_i = p.t; S/p$$
$$(\texttt{functor}_\varsigma(x_i : M)M')/p = \texttt{functor}_\varsigma(x_i : M)(M'/p(x_i)) \qquad (\texttt{val}_\ell\ v_i : \tau; S)/p = \texttt{val}_\ell\ v_i : \tau; S/p$$

Fig. 10. Module strengthening operation – M/p

$$\lfloor M \rfloor_b = M \qquad \lfloor M \rfloor_\varsigma = M$$
$$\lfloor \texttt{sig}\ S\ \texttt{end} \rfloor_\iota = \texttt{sig}\ \lfloor S \rfloor_\iota\ \texttt{end} \qquad \lfloor \texttt{functor}_\varsigma(x_i : M)M' \rfloor_\iota = \texttt{functor}_\varsigma(x_i : M)\lfloor M' \rfloor_\iota$$
$$\lfloor \varepsilon \rfloor_\iota = \varepsilon \qquad \lfloor \texttt{functor}(x_i : M)M' \rfloor_\iota = \texttt{functor}(x_i : \lfloor M \rfloor_\iota)\lfloor M' \rfloor_\iota$$
$$\lfloor \texttt{val}_\ell\ v_i : \tau; S \rfloor_\iota = \begin{cases} \texttt{val}_\iota\ v_i : \tau; \lfloor S \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor S \rfloor_\iota & \text{otherwise} \end{cases}$$
$$\lfloor \texttt{type}_\ell\ t_i = \tau; S \rfloor_\iota = \begin{cases} \texttt{type}_\iota\ t_i = \tau; \lfloor S \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor S \rfloor_\iota & \text{otherwise} \end{cases}$$
$$\lfloor \texttt{type}_\ell\ t_i; S \rfloor_\iota = \begin{cases} \texttt{type}_\iota\ t_i; \lfloor S \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor S \rfloor_\iota & \text{otherwise} \end{cases}$$
$$\lfloor \texttt{module}_m\ m_i : M; S \rfloor_\iota = \begin{cases} \texttt{module}_\iota\ m_i : \lfloor M \rfloor_\iota; \lfloor S \rfloor_\iota & \text{when } m \succ \iota \\ \lfloor S \rfloor_\iota & \text{otherwise} \end{cases}$$

Fig. 11. Module specialization operation – $\lfloor M \rfloor_m$

$$\frac{\sigma : [1;m] \to [1;n] \qquad \forall i \in [1;m],\ \Gamma; D_1; \ldots; D_n \blacktriangleright_m D_{\sigma(i)} <: D_i}{\Gamma \blacktriangleright_m (\texttt{sig}\ D_1; \ldots; D_n\ \texttt{end}) <: (\texttt{sig}\ D'_1; \ldots; D'_m\ \texttt{end})} \qquad \frac{m <: \ell_1 \succ \ell_2 \qquad \Gamma \triangleright_{\ell_2} \tau_1 \approx \tau_2}{\Gamma \blacktriangleright_m (\texttt{val}_{\ell_1}\ v_i : \tau_1) <: (\texttt{val}_{\ell_2}\ v_i : \tau_2)}$$

$$\frac{m <: m_1 \succ m_2 \qquad \Gamma \blacktriangleright_{m_2} M_1 <: M_2}{\Gamma \blacktriangleright_m (\texttt{module}_{m_1}\ x_i = M_1) <: (\texttt{module}_{m_2}\ x_i = M_2)} \qquad \frac{\Gamma \blacktriangleright_\ell M'_a <: M_a \qquad \Gamma, (\texttt{module}\ x : \mathrm{M}'_\mathrm{a})_\ell \blacktriangleright_\ell M_r <: M'_r}{\Gamma \blacktriangleright_\ell \texttt{functor}(x : M_a)M_r <: \texttt{functor}(x : M'_a)M'_r}$$

$$\frac{\Gamma \blacktriangleright_\varsigma M'_a <: M_a \qquad \Gamma, (\texttt{module}\ x : \mathrm{M}'_\mathrm{a})_\varsigma \blacktriangleright_\varsigma M_r <: M'_r}{\Gamma \blacktriangleright_\varsigma \texttt{functor}_\varsigma(x : M_a)M_r <: \texttt{functor}_\varsigma(x : M'_a)M'_r} \qquad \frac{m <: \ell_1 \succ \ell_2 \qquad \Gamma \triangleright_{\ell_2} \tau_1 \approx \tau_2}{\Gamma \blacktriangleright_m (\texttt{type}_{\ell_1}\ t_i = \tau_1) <: (\texttt{type}_{\ell_2}\ t_i = \tau_2)}$$

$$\frac{m <: \ell_1 \succ \ell_2}{\Gamma \blacktriangleright_m (\texttt{type}_{\ell_1}\ t_i) <: (\texttt{type}_{\ell_2}\ t_i)} \qquad \frac{m <: \ell_1 \succ \ell_2 \qquad \Gamma \triangleright_{\ell_2} t_i \approx \tau}{\Gamma \blacktriangleright_m (\texttt{type}_{\ell_1}\ t_i) <: (\texttt{type}_{\ell_2}\ t_i = \tau)} \qquad \frac{m <: \ell_1 \succ \ell_2}{\Gamma \blacktriangleright_m (\texttt{type}_{\ell_1}\ t_i = \tau_1) <: (\texttt{type}_{\ell_2}\ t_i)}$$

Fig. 12. Module subtyping rules – $\Gamma \blacktriangleright_m M <: M'$

type of the identifier towards the current location m. The specialization operation, which was described in Section 4.2, is noted $\lfloor M \rfloor_m$ and is defined in Figure 11.

There are two new typing rules compared to Leroy (1995): Rules MixedFunctor and MixedApplication define mixed functor definition and application. We use INJS(·) which returns the set of all injections in client declarations.

4.5 Subtyping and equivalence of modules

Subtyping rules are given in Figure 12. For brevity, we note $m <: m_1 \succ m_2$ as a shorthand for $m <: m_1 \wedge m <: m_2 \wedge m_1 \succ m_2$, that is, both m_1 and m_2 are valid locations for components of a module on location m and location m_1 encompass location m_2. Note that the following holds:

$$\Gamma \blacktriangleright_m \texttt{struct val}_b\ t_i : \mathit{int}\ \texttt{end} <: \texttt{struct val}_c\ t_i : \mathit{int}\ \texttt{end}$$

This is perfectly safe, since for any identifier x_i on base, $\texttt{let}_c\ x'_j = x_i$ is always valid. This allows programmers to declare some code on base (and get the guarantee that the code is only using usual OCaml constructs) but to expose it as client or server in the module type.

5 TARGET LANGUAGES

Eliom$_m$ is a compiled language that produces two programs: one server program (which is linked and executed on the server) and a client program (which is compiled to JavaScript and executed in a browser). Description of the complete compilation toolchain, including emission of JavaScript code, is out of scope of this article. Instead, we describe the compilation process in term of emission of client and server programs in an ML-like language equipped with additional communication primitives.

5.1 Base language

We note ML$_\varepsilon$ the simple ML language without any tierless features. This language uses the module language described in Leroy (1995) and the core language described in Wright and Felleisen (1994). Alternatively, the syntax and type system for this language can be obtained by ignoring the colored part in the definition of Eliom$_m$ in Section 3 and Section 4. We note $\Gamma \triangleright_{\text{ML}} e : \tau$; $\Gamma \blacktriangleright_{\text{ML}} m : M$; $\Gamma \vDash_{\text{ML}} \tau$ and $\Gamma \vDash_{\text{ML}} M$ the associated typing and well-formedness relations. Given an Eliom$_m$ module m, we note $m_{[\ell \mapsto \ell']}$ the substitution on locations. Given an ML$_\varepsilon$ module m, we note $m_{[\text{ML} \mapsto \ell]}$ the Eliom$_m$ module where all the module components have been annotated with location ℓ. We extend both notations to module types and environments.

Proposition 5.1. *Given an Eliom$_m$ module type M and a location $\ell \in \{b, c, s\}$, if $\Gamma \vDash_\ell M$, then $\lfloor M \rfloor_\ell = M$.*

Proof. By definition of $<:$, M can only contain declarations on ℓ. This means that, by reflexivity of $\succ$, only specialization rules that leave the declaration unchanged are involved. □

Proposition 5.2. *Given an Eliom$_m$ module type M and a location $\ell \in \{c, s\}$, if $\Gamma \vDash_b M$, then $\lfloor M \rfloor_\ell = M_{[b \mapsto \ell]}$.*

Proof. We remark that for all $\ell \in \{c, s\}$, $b \succ \ell$. Additionally, mixed functors cannot appear on base (since $\varsigma \not\succ b$). We can then proceed by induction over the rules for specialization. □

Proposition 5.3. *Given ML$_\varepsilon$ type τ, expression e, module m and module type M and locations ℓ, ℓ':*

$$\Gamma \vDash_{ML} \tau \implies \Gamma_{[ML \mapsto \ell']} \vDash_\ell \tau \qquad \text{Where } \ell' \succ \ell$$

$$\Gamma \triangleright_{ML} e : \tau \implies \Gamma_{[ML \mapsto \ell']} \triangleright_\ell e : \tau \qquad \text{Where } \ell' \succ \ell$$

$$\Gamma \vDash_{ML} M \implies \Gamma_{[ML \mapsto \ell']} \vDash_\ell M_{[ML \mapsto \ell]} \qquad \text{Where } \ell' \succ \ell$$

$$\Gamma \blacktriangleright_{ML} m : M \implies \Gamma_{[ML \mapsto \ell']} \blacktriangleright_\ell m_{[ML \mapsto \ell]} : M_{[ML \mapsto \ell]} \qquad \text{Where } \ell' \succ \ell$$

Proof. We remark that each syntax, typing rule or well formedness rule for ML_ε has a direct equivalent rule in Eliom$_m$. We can then simply rewrite the proof tree of the hypothesis to use the Eliom$_m$ type and well-formedness rules. We consider only some specific cases:

- By Proposition 5.1 and since the modules are of uniform location, the specialization operation in Var and ModVar are the identity.
- The side conditions $\ell' <: \ell$ are always respected since the modules are of uniform location and by reflexivity of $<:$.
- The side conditions $\ell' > \ell$ are respected by hypothesis. □

Proposition 5.4. *Given ML_ε type τ, expression e, module m and module type M:*

$$\Gamma \vDash_b \tau \implies \Gamma_{[b\mapsto ML]} \vDash_{ML} \tau \qquad \Gamma \vDash_b M \implies \Gamma_{[b\mapsto ML]} \vDash_{ML} M_{[b\mapsto ML]}$$
$$\Gamma \triangleright_b e : \tau \implies \Gamma_{[b\mapsto ML]} \triangleright_{ML} e : \tau \qquad \Gamma \blacktriangleright_b m : M \implies \Gamma_{[b\mapsto ML]} \blacktriangleright_{ML} m_{[b\mapsto ML]} : M_{[b\mapsto ML]}$$

Proof. We first remark that the following features are forbidden in the base part of the language: injections, fragments, mixed functors and any other location than base. The rest of the language contains no tierless features and coincide with ML_ε. We can then proceed by induction over the proof trees. □

By Propositions 5.3 and 5.4, we can completely identify the language ML_ε and the part of Eliom$_m$ on the base location b. This is of course by design: the base location allows us to reason about the host language, OCaml, inside the new language Eliom$_m$. It also provides the guarantee that anything written in the base location does not contain any communication between client and server. Furthermore, it means that, given a file previously typechecked by an ML_ε typechecker, we can directly use the module types either on base, but also on the client or on the server, by simply annotating all the signature components. In the rest of this article, we omit location substitutions of the form $_{[\text{ML}\mapsto b]}$ and $_{[b\mapsto\text{ML}]}$.

5.2 ML_S and ML_C

We introduce the two target languages ML_C and ML_S as extension of ML_ε. The additions in these two new languages are highlighted in Figure 13. Typing is provided in Figure 14. The semantics of most of those primitives are detailed in Radanne et al. (2016a). We simply give an informal reminder. We introduce a new class of identifiers, called "references" and noted in bold: $\mathbf{v}$. Those identifiers are dynamically created and have no scoping rules. They are dynamically bound, as opposed to the statically bound identifiers. In practice, they are implemented with uniquely generated names and associative tables. References are used to synchronize values between the server and the client. A reference used inside an expression is always of type `serial`.

5.2.1 Injections. For an injection, we associate server-side the injected value e to a reference $\mathbf{v}$ using the construction `injection` $\mathbf{v}$ e, where e is of type `serial`. When the server execution is done, a mapping from references to injected values is sent to the client. $\mathbf{v}$ is then used client-side to access the value.

5.2.2 Fragments. To implement fragments, we use two primitives:

- `bind` $\mathbf{f} = e$ declares a client function.

- `fragment` **f** e is a delayed application function that is used on the server to register that, on the client, the function associated to **f** will be applied to the arguments e. All the arguments must be of type `serial`. It returns a value of type `frag`, which holds a unique identifier refering to the result of this application..

5.2.3 Converters. For each converter f, we note f^s and f^c the server side encoding function and the client side decoding function. If f is of type $\tau_s \rightsquigarrow \tau_c$, then f^s is of type $\tau_s \to$ `serial` and f^c is of type `serial` $\to \tau_c$.

5.2.4 Modules. We introduce three new module-related construction.

- `bind`$_m$ **F** $= m$ is equivalent to `bind` for modules. It is a client instruction that associates the module or functor m to the reference **F**. This construction cannot be nested. We denote by $\mathbf{F}(\overline{\mathbf{p}}).\mathbf{f}$ references that are created by `bind` inside a functor bound by `bind`$_m$.
- `fragment`$_m$ **F** $(\overline{\mathbf{x}})$ is equivalent to `fragment` **f** e for modules. It is a delayed functor application that is used on the server to register that the functor associated to **F** will have to be applied to the modules associated to **x**. It returns a fresh reference that represents the resulting module. Contrary to `fragment`, it can only be applied to module references.
- p.dyn returns a reference that represents the client part of a server module p. This is used for Eliom$_m$ mixed structure that have both a server and a client part.

From a practical point of view, these new primitives are implemented using first-class modules and the primitives `bind` and `fragment` respectively.

ML$_s$ grammar

e ::= ... | `fragment` **f** $\overline{e}$ (Fragment call)
τ ::= ... | `frag` (Fragment type)
d ::= ...
| `module dyn =` `fragment`$_m$ **F** $\overline{(p.\mathsf{dyn})}$
| `injection` **v** e (Injection)
| `end ()` (End token)

ML$_c$ grammar

e ::= ... | **v** (Reference)
d ::= ...
| `bind` **f** $= e$ (Fragment closure)
| `bind`$_m$ **F** $= m$ (Module fragment closure)
| `exec ()` (Fragment execution)

Fig. 13. Grammar for ML$_s$ and ML$_c$ as extensions of ML$_\varepsilon$

ML$_s$ Typing rules

$$\frac{\forall i,\ \Gamma \triangleright_{\mathrm{ML_s}} e_i : \mathtt{serial}}{\Gamma \triangleright_{\mathrm{ML_s}} \mathtt{fragment}\ \mathbf{f}\ \overline{e_i} : \mathtt{frag}} \qquad \frac{\Gamma \triangleright_{\mathrm{ML_s}} e : \mathtt{serial}}{\Gamma \blacktriangleright_{\mathrm{ML_s}} \mathtt{injection}\ \mathbf{v}\ e : \varepsilon}$$

$$\frac{}{\Gamma \blacktriangleright_{\mathrm{ML_s}} \mathtt{end}\ () : \varepsilon}$$

$$\frac{}{\Gamma \blacktriangleright_{\mathrm{ML_s}} \mathtt{module\ dyn} = \mathtt{fragment}_m\ \mathbf{F}\ \overline{(p_i.\mathsf{dyn})}\mathsf{dyn} : \varepsilon}$$

ML$_c$ Typing rules

$$\frac{\Gamma \triangleright_{\mathrm{ML_c}} e : \tau}{\Gamma \blacktriangleright_{\mathrm{ML_c}} \mathtt{bind}\ \mathbf{f} = e : \varepsilon} \qquad \frac{}{\Gamma \triangleright_{\mathrm{ML_c}} \mathbf{v} : \mathtt{serial}}$$

$$\frac{\Gamma \blacktriangleright_{\mathrm{ML_c}} m : M}{\Gamma \blacktriangleright_{\mathrm{ML_c}} \mathtt{bind}_m\ \mathbf{F} = m : \varepsilon} \qquad \frac{}{\Gamma \blacktriangleright_{\mathrm{ML_c}} \mathtt{exec}\ () : \varepsilon}$$

Fig. 14. Additional types rules for ML$_c$ and ML$_s$

6 COMPILATION

Compilation for Eliom is the process of transforming *one* Eliom$_m$ program into two distinct ML$_S$ and ML$_C$ programs. Before giving a more formal description, we present the compilation process with three examples of increasing complexity that demonstates the most distinctive features of the language.

Our first example, in Example 4, only contains simple declarations involving fragments and injections without modules. In this example, a first fragment is created. It only contains an integer and is bound to a. A second fragment that uses a is created and bound on the server to b. Finally, b is used on the client via an injection. The program returns 4. For each fragment, we have a `bind` declaration. The client expression contained in the fragment is abstracted and transformed in a closure that is bound to a fresh reference. The number of arguments of the closure corresponds to the number of injections inside the fragment. On the server, each fragment is replaced by a call to the primitive `fragment`. The argument of the call are the identifier of the closures and all the injections that are contained in the fragment. This primitive registers that the closure should be executed later on. Since all the arguments of `fragment` should be of type `serial`, we apply the client and server parts of the converters at the appropriate places. The `exec` and `end` primitives synchronize the execution so that the order of side effects is preserved. When `exec` is encountered, it executes queued fragment up to an `end` token which was pushed by an `end` primitive. Note that injections, which occur outside of fragments, and escaped values, which occur inside fragments, are compiled in very different way. Injections have the useful property that the use site and number of injections is completely static: we can collect all the injections on the server, independently of the client control flow and send them to the client. This is the property that allows us to avoid communications from the client to the server.

Eliom$_m$	ML$_S$	ML$_C$
$\texttt{let}_s\ a = \{\{\ 1\ \}\};$	$\texttt{let}\ a = \texttt{fragment}\ \mathbf{f_0}\ ();$ $\texttt{end}\ ();$	$\texttt{bind}\ \mathbf{f_0} = \lambda().1;$ $\texttt{exec}\ ();$
$\texttt{let}_s\ b = \{\{\ \texttt{frag\%}a + 1\ \}\};$	$\texttt{let}\ b = \texttt{fragment}\ \mathbf{f_1}\ (\texttt{frag}^s\ a);$ $\texttt{end}\ ();$	$\texttt{bind}\ \mathbf{f_1} = \lambda v.((\texttt{frag}^c\ v) + 1);$ $\texttt{exec}\ ();$
$\texttt{let}_c\ res = \texttt{frag\%}b + 2;$	$\texttt{injection}\ \mathbf{v}\ (\texttt{frag}^s\ b);$	$\texttt{let}\ res = (\texttt{frag}^c\ \mathbf{v}) + 2;$

Example 4. Compilation of expressions

We now present an example with client and server modules in Example 5. We declare a server module containing a client fragment, a client functor containing an injection, a client functor application and finally the client return value, with another injection. The important thing to note here is that for client and server modules or functors, the special instructions for fragments and injection can be freely lifted to the outer scope. Indeed, let us consider the injection inside the client functor f: the functor argument must be on the client, hence it cannot introduce new server binding. As such, the server identifier that is injected must have been introduced outside of functor, which means it can be lifted. A similar remark can be made for client fragments in server modules.

Finally, Example 6 presents the compilation of mixed modules. In this example, we create a mixed structure x containing a server declaration and a client declaration with an injection. We define a functor f that takes a module containing a client integer and use it both inside a client fragment, and inside a client declaration. We then apply f to x and use an injection to compute the final result of the program.

The slicing of the mixed module x is similar to the procedure for programs: we treat each declaration and use the `injection` primitive as needed. Additionally, we introduce a new identifier $\mathbf{F_0}$ that allows us to associate the server part of the module to the corresponding client part. In ML$_S$ we define the `dyn` field using the `fragment`$_m$ primitive that returns said identifier. In ML$_C$, the `bind`$_m$ primitive allows to associate the identifier to the module in a table. For

ELIOM$_m$	ML$_S$	ML$_C$

module$_s$ x = struct
  let$_s$ a = {{ 2 }}
  let$_s$ b = 4
end;
module$_c$ $f(x : M)$ = struct
  let$_c$ $a = x.b +$ int%$x.b$
end;
module$_c$ $y = f$(struct let$_c$ $b = 2$ end);
let$_c$ res = frag%$x.a + y.a$;
	module x = struct
	  let a = fragment $\mathbf{f_0}$ ()
	  let $b = 4$
	end; end ();
	
	injection $\mathbf{v_0}$ (int^s $x.b$);
	
	injection $\mathbf{v_1}$ (frag^s $x.a$);
		bind $\mathbf{f_0} = \lambda().2$;
		
		exec ();
		module $f(x : M)$ = struct
		  let $a = x.b +$ (int^c $\mathbf{v_0}$)
		end;
		module $y = f$(struct let $b = 2$ end);
		let res = (frag^c $\mathbf{v_1}$) + $y.a$;

Example 5. Compilation of client and server modules and functors

functors, each call to fragment$_m$ generates a new, fresh identifier. On the client, we emit two different functors: one that contains only the client declarations to be used inside the rest of the client code and one that contains both client declaration and also calls to the bind and exec primitives. This module is used to perform client side effects. When the server version of f is applied, a call to $\mathbf{F_1}$ is registered and will be executed when the client reaches the associated exec call (here, the last one).

6.1 Slicing rules

The slicing rules are defined in Figure 15. Given an ELIOM$_m$ module m or module type M, and a location ι that is either client or server, we note $\langle m \rangle_\iota$ the result of the compilation of m to the location ι. The result of $\langle m \rangle_s$ is a module of ML$_S$ and the result of $\langle m \rangle_c$ is a module in ML$_C$.

The compilation rules for the core language are similar to the one presented by Radanne et al. (2016a). As before, we use the notation $e\,[a \mapsto b]$ to denote the substitution of a by b in e. $e\,[a_i \mapsto b_i]_i$ denotes the repeated substitution of a_i by b_i in e. We note FRAGS(e) (resp. INJS(e)) the fragments (resp. injections) present in the expression e. We note $\overline{e_i}$ the list of elements e_i. For ease of presentation, we use d_m and D_m for definitions and declarations located on location m.

Our slicing rules only applies to *sliceable* programs. A program is said sliceable if mixed structures are only defined at top level, or directly inside a toplevel mixed functor. We explain how to partially relax this restriction in Section 6.4. We also require the program to be well typed.

ELIOM$_m$	ML$_S$	ML$_C$

module$_\varsigma$ x = struct
  let$_s$ $a = 2$
  let$_c$ $b = 4 +$ int%a
end;
 	
module$_\varsigma$ $f(y : M)$ = struct
  let$_s$ c = {{ $y.b$ }}
  let$_c$ $d = 2 * y.b$
end;
 	
module$_\varsigma$ $z = f(x)$;
let$_c$ res = frag%$z.c + z.d$;
	module x = struct
	  module dyn = fragment$_m$ $\mathbf{F_0}$ ();
	  let $a = 2$; end ();
	  injection $\mathbf{v_0}$ (int^s a);
	end;
	module $f(y : M)$ = struct
	  module dyn = fragment$_m$ $\mathbf{F_1}$ (y.dyn);
	  let c = fragment dyn.$\mathbf{f_0}$ (); end ();
	end;
	
	module $z = f(x)$; end ();
	injection $\mathbf{v_1}$ (frag^s $z.c$);
		module x = struct
		  let $b = 4 +$ (int^c $\mathbf{v_0}$)
		end;
		bind$_m$ $\mathbf{F_0}() = x$;
		module $f(y : M)$ = struct
		  let $d = 2 * y.b$
		end;
		bind$_m$ $\mathbf{F_1}(y : M)$ = struct
		  bind $\mathbf{f_0} = \lambda().(y.b)$; exec ();
		  let $d = 2 * y.b$
		end;
		module $z = f(x)$;exec ()
		let$_c$ res = (frag^c $\mathbf{v_1}$) + $z.d$;

Example 6. Compilation of a mixed functor

We now describe how to slice the various constructions of our language. Base structure and signature components are kept untouched. Indeed, according to Theorem 5.4, base elements are valid ML_ε elements. We do not need to modify them in any way. Signature components that are not valid on the target location are simply omitted. Signature components that are valid on the target have their type expressions translated. The translation of a type expression to the client is the identity: indeed, there are no new Eliom$_m$ type constructs that are valid on the client. Server types, on the other hand, can contains pieces of client types inside fragments $\{\tau_c\}$ and inside converters $\tau_s \rightsquigarrow \tau_c$. Fragments in ML_S are represented by a primitive type, `fragment`, without parameters. The type of converters is represented by the type of their server part, which is $\tau_s \to$ `serial`. Module and module type expressions are traversed recursively. Functors and functor applications have each part sliced. Mixed functors are turned into normal functors.

Slicing of structure components inserts additional primitives that were described in Section 5.2. In client structure components, we need to handle injections. We associate each injection to a new fresh reference noted **v**. In ML_S, we use the `injection` primitive to register the fact that the given server value should be associated to a given reference. In ML_C, we replace each injection by its associated reference. This substitution is applied both inside expressions and structures. Note that for each injection $f\%x$, we use the encoding part f^s and decoding part f^c for the server and client code, respectively. For server structures components, we apply a similar process to handle fragments. For each fragment, we introduce a reference noted **f**. In ML_S, we replace each fragment by a call to `fragment` with argument the associated reference and each escaped value inside the fragment (with the encoding part of the converters). We also add, after the translated component, a call to `end` which indicates that the execution of the component is finished. In ML_C, we use the `bind` primitives to associate to each reference a closure where all the escaped values are abstracted. We also introduce the decoding part of each converter for escaped values. We then call `exec`, which executes all the pending fragments until the next `end` (). This allows to synchronize interleaved side effects between fragments and client components.

Given the constraint of sliceability, a mixed module is either a multi-argument functor returning a structure, or it does not contain any structure at all. Mixed modules without structures can simply be sliced by leaving the module expression unchanged. Mixed module types are also straightforward to slice. Mixed structures (with an arbitrary number of arguments) need special care. For each structure, we generate a fresh reference noted **F**. In ML_S, we introduce a new field in the structure called `dyn`. The value of this field is the result of a call to the primitive `fragment`$_m$ with arguments **F** and all the `dyn` fields of the arguments of the functor. In ML_C, we create two structures for each mixed structure. One is simply a client functor where all the server parts have been removed. Note here that we don't use the slicing operation. The resulting structure does not contain any call to `bind` and `exec`. We also create another structure that uses the regular slicing operation. This structure is associated to **F** with the `bind`$_m$ primitive

6.2 Notes about mixed functors

Mixed functors can be seen from two perspectives. The client part of a mixed functor is exactly the same as a pure client functor. Indeed, since we disallow arbitrary injections, we can remove all the server code and leave the client part untouched. The server part, however, can be seen as a pair of a server module and an client module hidden inside a fragment. Indeed, given the presence of fragments inside the server part, each mixed functor application done on the server needs a corresponding functor application on the client which performs all the client-side evaluation. This means that we need a way to associate each server version of mixed modules to its client version. In order to do that,

Signatures

$$\begin{aligned}\langle D_b; S\rangle_\iota &= D_b; \langle S\rangle_\iota \\ \langle D_m; S\rangle_\iota &= \langle S\rangle_\iota && \text{when } m \not\succ \iota \\ &= \langle D_m\rangle_\iota ; \langle S\rangle_\iota && \text{when } m \succ \iota\end{aligned}$$

Type expressions

$$\begin{aligned}\langle\{\tau\}\rangle_s &= \texttt{frag} \\ \langle \tau_s \rightsquigarrow \tau_c\rangle_s &= \langle\tau_s\rangle_s \to \texttt{serial}\end{aligned}$$

Module Type Expressions

$$\begin{aligned}\langle \texttt{sig}\ s\ \texttt{end}\rangle_\iota &= \texttt{sig}\ \langle s\rangle_\iota\ \texttt{end} \\ \left\langle \texttt{functor}(x_i : M) M'\right\rangle_\iota &= \texttt{functor}(x_i : \langle M\rangle_\iota)\left\langle M'\right\rangle_\iota \\ \left\langle \texttt{functor}_\varsigma(x_i : M) M'\right\rangle_\iota &= \texttt{functor}(x_i : \langle M\rangle_\iota)\left\langle M'\right\rangle_\iota\end{aligned}$$

Declarations and Definitions

$$\begin{aligned}\langle \texttt{type}_\iota\ t_i = \tau\rangle_\iota &= \texttt{type}\ t_i = \langle\tau\rangle_\iota \\ \langle \texttt{type}_\iota\ t_i\rangle_\iota &= \texttt{type}\ t_i \\ \langle \texttt{val}_\iota\ v_i : \tau\rangle_\iota &= \texttt{val}\ v_i : \langle\tau\rangle_\iota \\ \langle \texttt{let}_\iota\ v_i = e\rangle_\iota &= \texttt{let}\ v_i = e \\ \langle \texttt{module}_m\ x_i : M\rangle_\iota &= \texttt{module}\ x_i : \langle M\rangle_\iota \\ \langle \texttt{module}_\iota\ x_i = m\rangle_\iota &= \texttt{module}\ x_i = \langle m\rangle_\iota\end{aligned}$$

Module Expressions

$$\begin{aligned}\langle \texttt{struct}\ s\ \texttt{end}\rangle_\iota &= \texttt{struct}\ \langle s\rangle_\iota\ \texttt{end} \\ \left\langle m(m')\right\rangle_\iota &= \langle m\rangle_\iota\left(\left\langle m'\right\rangle_\iota\right) \\ \langle \texttt{functor}(x_i : M) m\rangle_\iota &= \texttt{functor}(x_i : \langle M\rangle_\iota)\langle m\rangle_\iota \\ \left\langle \texttt{functor}_\varsigma(x_i : M) m\right\rangle_\iota &= \texttt{functor}(x_i : \langle M\rangle_\iota)\langle m\rangle_\iota\end{aligned}$$

Structure components

$$\langle d_b; s\rangle_\iota = d_b; \langle s\rangle_\iota \qquad \left\langle d_\varsigma; s\right\rangle_\iota = \left\langle d_\varsigma\right\rangle_\iota ; \langle s\rangle_\iota$$

$$\langle d_c; s\rangle_s = \left[\texttt{injection}\ \mathbf{v_i}\ (f_i^s\ v_i);\right]_i \langle s\rangle_s$$

$$\langle d_c; s\rangle_c = \begin{array}{l} d_c\left[\overline{f_i \% v_i} \mapsto \overline{(f_i^c\ \mathbf{v}_i)}\right]; \\ \langle s\rangle_c \end{array}$$

Where $\begin{cases}\overline{f_i \% v_i} = \text{INJS}(d_c) \\ \overline{\mathbf{v}_i} \text{ is a list of fresh variables.}\end{cases}$

$$\langle d_s; s\rangle_s = \begin{array}{l} \langle d_s\rangle_s \left[\{\{\ e_i\ \}\} \mapsto \texttt{fragment}\ \mathbf{f_i}\ \overline{(f_i^s\ a)}_i\right]_i; \\ \texttt{end}\ (); \\ \langle s\rangle_s \end{array}$$

$$\langle d_s; s\rangle_c = \begin{array}{l} \left[\texttt{bind}\ \mathbf{f}_i = \lambda(\overline{x})_i.(e_i\left[\overline{(f\%a)}_i \mapsto \overline{(f_i^c\ x)}_i\right]);\right]_i \\ \texttt{exec}\ (); \\ \langle s\rangle_c \end{array}$$

Where $\begin{cases}\overline{\{\{\ e_i\ \}\}} = \text{FRAGS}(d_s) \\ \overline{\mathbf{f}_i} \text{ is a list of fresh variables.} \\ \forall i, \overline{(f\%a)}_i = \text{INJS}(e_i) \\ \forall i, (\overline{x})_i \text{ is a list of fresh variables;}\end{cases}$

Mixed structures

$$\left\langle \texttt{module}_\varsigma\ x_i\overline{(a_{i_k} : M_k)} = \texttt{struct}\ s\ \texttt{end}\right\rangle_s = \begin{array}{l} \texttt{module}\ x_i\overline{(a_{i_k} : \langle M_k\rangle_c)} = \texttt{struct} \\ \quad \texttt{module dyn} = \texttt{fragment}_m\ \mathbf{F_i}\ \overline{(a_{i_k}.\texttt{dyn})}; \\ \quad \langle s\rangle_s \\ \texttt{end} \end{array}$$

$$\left\langle \texttt{module}_\varsigma\ x_i\overline{(a_{i_k} : M_k)} = \texttt{struct}\ s\ \texttt{end}\right\rangle_c = \begin{array}{l} \texttt{module}\ x_i\overline{(a_{i_k} : \langle M_k\rangle_c)} = \texttt{struct}\ s'|_c\ \texttt{end}; \\ \texttt{bind}_m\ \mathbf{F_i}\overline{(a_{i_k} : \langle M_k\rangle_c)} = \texttt{struct}\ \langle s\rangle_c\ \texttt{end}; \end{array}$$

Where $\mathbf{F_i}$ is a fresh reference.

Mixed modules

$$\left\langle \texttt{module}_\varsigma\ x_i = m\right\rangle_\iota = \texttt{module}\ x_i = \langle m\rangle_\iota \qquad \left\langle \texttt{module}_\varsigma\ x_i : M\right\rangle_\iota = \texttt{module}\ x_i : \langle M\rangle_\iota$$

Fig. 15. Slicing – $\langle\cdot\rangle_\iota$

we equip the server version with a new field dyn which contains a unique identifier. This identifier is later transmitted to the client and used to retrieve the client part of the mixed module.

In order to handle functors, we register each mixed functor application that happens on the server and use dyn fields to perform them client side. Since mixed functors can only be applied to mixed structures, they necessarily have a dyn field.

The design space for mixed functors is quite large. We could consider only *fully separable* functors: mixed functors such that client and server execution are completely independent. While this would be easy to implement, it would also mean preventing any meaningful usage of fragments inside functors. Our version of mixed functors is slightly more expressive: the client part of the functor is indeed independent from the server part, but the server part is not. The cost is that we must do some extra book-keeping to ensure that for each server-side application of a mixed functors, all the client side effects are performed. We believe this expressive power is sufficient for most use cases. In particular, it is sufficient for converters, which are described in Section 7.4.

6.3 Results on slicing

One desirable property is that the introduction of new elements in the language and the slicing operation does not compromise the guarantees provided by the host language. To ensure this, we show that slicing a well typed Eliom$_m$ program provides two well typed ML$_c$ and ML$_s$ programs.

We only consider typing environments Γ such that $(\texttt{frag})_s \in \Gamma$ and $(\texttt{serial})_b \in \Gamma$. We also extend the slicing operation to typing environments. Slicing a typing environments is equivalent to slicing a signature with additional rules for converters. Converters, in Eliom$_m$, are not completely first class: they are only usable in injections and not manipulable in the expression language. As such, they must be directly provided by the environment. We add the two following slicing rules that ensures that converters are properly present in the sliced environment:

$$\langle \texttt{val } f : \tau_s \rightsquigarrow \tau_c \rangle_s = \texttt{val } f^s : \langle \tau_s \rangle_s \rightsquigarrow \texttt{serial} \qquad \langle \texttt{val } f : \tau_s \rightsquigarrow \tau_c \rangle_c = \texttt{val } f^c : \texttt{serial} \rightsquigarrow \tau_c$$

Theorem 6.1 (Slicing preserves typing). *Let us consider m and M such that $\Gamma \blacktriangleright_\varsigma m : M$.*
Then $\langle \Gamma \rangle_s \blacktriangleright \langle m \rangle_s : \langle M \rangle_s$ and $\langle \Gamma \rangle_c \blacktriangleright \langle m \rangle_c : \langle M \rangle_c$

Proof. We proceed by induction over the proof tree of $\Gamma \blacktriangleright_\varsigma m : M$. The only difficult cases are client and server structure components and mixed structures. For brevity, we only detail the case of client structure components with one injection.

Let us consider d_c such that $\Gamma \blacktriangleright_\varsigma d_c; s : S$ and $\text{INJS}(d_c) = f\%v$. We note $\mathbf{v}$ the fresh reference. By definition of the typing relation on Eliom$_m$, there exists Γ' and τ_c, τ_s such that $\Gamma \subset \Gamma'$, $\Gamma' \triangleright_s f : \tau_s \rightsquigarrow \tau_c$ and $\Gamma' \triangleright_s v : \tau_s$. We observe that there cannot be any server bindings in d_c, Hence we can choose $\Gamma' = \Gamma$.

By definition of slicing on typing environments, $(f^s : \langle \tau_s \rangle_s \to \texttt{serial}) \in \langle \Gamma \rangle_s$ and $(f^c : \texttt{serial} \to \tau_c) \in \langle \Gamma \rangle_c$. By definition of ML$_c$ and ML$_s$ typing rules, we have $\langle \Gamma \rangle_s \triangleright_{\text{ML}_s} (f^s\ v) : \texttt{serial}$ and $\langle \Gamma \rangle_c \triangleright_{\text{ML}_c} (f^c\ \mathbf{v}) : \tau_c$.

We easily have that $\langle \Gamma \rangle_s \blacktriangleright_{\text{ML}_s} \texttt{injection } \mathbf{v}\ (f^s\ v) : \varepsilon$.

By induction hypothesis on $\Gamma, (v_j : \tau_c)_c \triangleright_\varsigma d_c \left[f\%v \mapsto v_j \right] : \varepsilon$ where v_j is fresh, we have $\langle \Gamma \rangle_c, (v_j : \tau_c) \triangleright_{\text{ML}_c} d_c \left[f\%v \mapsto v_j \right] : \varepsilon$. We can then replace the proof tree of v_j by the one of $(f^c\ \mathbf{v})$. We simply need to ensure that the environments coincide. This is the case since f^c cannot be introduced by new bindings. We can then remove the binding of v_j from the environment, since it is unused. We obtain that $\langle \Gamma \rangle_c \triangleright_{\text{ML}_c} d_c \left[f\%v \mapsto (f^c\ \mathbf{v}) \right] : \varepsilon$ which allows us to conclude. □

6.4 Structure lifting

The sliceability requirement is quite restrictive. A set of rewriting rules can be used to increase the set of accepted programs. An example is given in Example 7. The idea is quite simple: we can lift mixed structures out of mixed structures by progressively moving their definition upward and out of a structure. For this, we need to add new functor arguments for each useful declaration. Since all mixed modules can be argument of mixed functors, we can only use mixed declarations. This procedure, however, is not complete.

```
module%mixed F (A : S) = struct
  module%mixed C = G(A.C)
  module%mixed M = struct
    let%client c = C.v
    let%server s = A.s
  end
end
```

(a) A mixed structure before lifting

```
module%mixed M' (A:S) (C':sig ... end) = struct
  let%client c = C'.C.v
  let%server s = A.s
end
module%mixed F (A : S) = struct
  module%mixed C = G(A.C)
  module%mixed M = M'(A)(C)
end
```

(b) A lifted mixed structure

Example 7. Mixed structure lifting

7 EXTENSIONS

We present several extension to Eliom$_m$. We give a quick informal sketch of how they might work.

7.1 Parameterized datatypes

One very useful tool in ML languages is the ability to define parameterized datatypes. For example `'a list` is the type of lists that contain elements of type `'a`. It is very natural to extend an ML language with declarations of parameterized datatypes inside structures and signatures: $\texttt{type}_\ell\ (\alpha_0, \ldots, \alpha_n)t = \tau$

We also would like *abstract* parameterized datatypes. However, a naive implementation would cause issues. Let us consider the following declaration:

$$(\texttt{struct type}_s\ \alpha\ t = \mathit{int} * \{\alpha\}\ \texttt{end}) : (\texttt{sig type}_s\ \alpha\ t\ \texttt{end})$$

The type variable α should only be instantiated with *client* types. This problem is very similar to the interaction between variance and abstract parameterized datatypes. The solution is simply to annotate type variables with a location. During checking of a type expression, we need to change the scoping location of type parameters, according to the definition of t.

7.2 Expressions inside injections

It is desirable to handle expressions inside injections, for example : $\texttt{let}_c\ y = f\%(x + 1)$. This can be implemented by hoisting the server expression to the nearest outer mixed scope. We however need to check proper scoping of bound identifiers to forbid client expressions of the form: $\lambda x.f\%\{\{\ x\ \}\}$. This also allows converters to be expressions.

7.3 Inference of mixed annotations

In Eliom$_m$, we present four annotations on modules: **b**ase, **s**erver, **c**lient and ς, mixed. We showed that base corresponds exactly to the underlying ML language. From a programming point of view, we can simply omit such annotations. Most mixed annotations can also be inferred: when typechecking a module declaration $\texttt{module}_m\ x = m$ where m is neither

client nor server, it might be either mixed or base. Note that any valid base structure is also a valid mixed structure. The main point is then to decide if a given module expression can be declared on base or not. The module expression m can be declared on base if all of the following properties hold:

- all the identifiers used in m are declared on base;
- if m contains a structure, all the definitions must be on base;
- there are no mixed functors.

We can infer mixed functors in a similar way by looking at the argument signature and the result module. Some modules can still be ambiguous, in particular in the presence of high order functors. However, this procedure allows to omit most "obvious" mixed and base annotations.

7.4 Converters

In our formalization, converters are global constants that are provided by the typing environment. We simply require them to have a client and a server part. The actual implementation and definition of new converters is not treated. Our current implementation relies on runtime dispatch of converters by inspecting the value.

However, one promising lead to implement converters in a way that is both convenient and more in line with the formalization is to use mixed modules following the signature shown in Figure 16. We can then use ad-hoc polymorphism with modular implicits (White et al. 2014). This is one of the main motivations for mixed functors: converters for parameterized datatypes would then be functors that take and return modules of type `CONV`.

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

Fig. 16. A signature for converters

8 IMPLEMENTATION

We implemented our extension of OCaml as a patch on the OCaml compiler and typechecker, including annotations on parametrized datatypes and inference of mixed annotations. Details on the implementation for the expression language can be found in Radanne et al. (2016b).

Our implementation is quite faithful to the formalization of the new module system. The main difference with Eliom$_m$ is that locations are annotated on identifiers instead of binders. The downside is that the specialization operation is more invasive (it needs to explore type expressions). However, it allows us to leave the implementation of the typing environment mostly untouched (no need to annotate bindings in the environment itself). This was considered preferable, as the implementation of typing environments in the OCaml typechecker is very complex. Another peculiarity of our implementation, which corresponds closely to our compilation scheme, is that each `.eliom` file is compiled to one `.cmi` file (the typing signature of the module) but two `.cmo` files (the output of the compiler, similar to `.o` for C). Finally, our implementation does not modify the representation of compiled objects. This means that any `.cmi` or `.cmo` compiled with the vanilla OCaml compiler can be used directly in Eliom, along with a number of tools. It is also possible to specify that a given pure-OCaml module can be loaded either as base, as client or as server.

9 RELATED WORK

A comprehensive comparison of the tierless expression language can be found in Radanne et al. (2016b). It is notoriously delicate to compare modules systems. Instead, we focus on the modularity and abstraction aspects. In particular,

separate compilation and data abstraction. Within these criteria, the various approaches can be separated into various categories: slicing as a global compiler transformation, interpreted languages and modular compiled languages.

9.1 Global slicing

One approach for slicing a tierless program into a client part and a server part is to apply a whole-program transformation over the complete program. Such approach is, by essence, incompatible with separate compilation. Furthermore, whole-program slicing usually rely on some other program transformations (inlining, monomorphisation, defunctorisation, ...) that tend to be non-modular and cross abstraction boundaries.

Ur/Web (Chlipala 2015a,b) is a statically typed ML-like tierless programming language. It only provides compilation units, not modules. Its approach to compilation is similar to MLton (MLton 2014): it applies a set of whole-program optimizations to remove all high order calls, then slice the program. This process is incompatible with separate compilation.

There has been several work on bringing static slicing to JavaScript (Chong et al. 2007; Philips et al. 2014). These approaches do not provide any tools to talk about modules and are whole-program transformations. Furthermore, JavaScript modules do not provide any form of data abstraction.

9.2 Dynamic slicing

Some interpreted languages rely on slicing at runtime to extract the client part of the program and send it alongside the generated Web page. While this is more expressive, it does not provide any of the guarantees provided by static slicing.

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. There is no static typing, JavaScript modules do not provide any data abstraction feature and the slicing is not modular.

Meteor.js (Meteor.js 2017) is a framework where both the client and the server side of an application are written in JavaScript. As a JavaScript extension, it also does not provide static typing nor any form of abstraction.

Links (Cooper et al. 2006) is an experimental functional language for client-server Web programming with a type and effect system. The slicing is type-directed, leveraging effects to annotate client, server or database functions. The current implementation of Links is interpreted and relies on dynamic slicing. It does not have a module system. Some work has been done on introducing static compilation (Cheney et al. 2013), but it relies on normalization by evaluation, which is not immediately compatible with separate compilation.

9.3 Modular languages

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. It inherits the Haskell features in term of modules and data abstraction. Furthermore, the tierless compiler for Haste relies heavily on the GHC, providing support for separate-compilation. However, a complete expressive module language for Haskell is still work in progress (Kilpatrick et al. 2014).

MetaOCaml (Kiselyov 2014) is an extension of OCaml for staged meta-programming. While the expression language is quite similar to the one in Eliom, MetaOCaml provides no support for modules. Staging annotations are only on expressions, not on declarations. Code generation and checking of the generated code is dynamic.

Modular macros (Nicole 2016; Yallop and White 2015) are another extension of OCaml. It uses staging to implement macros. It provides both a quotation-based expression language along with staging annotations on declarations. It also

aims to support modules and functors. Contrary to Eliom, there is only one type universe. Furthermore, the slicing can also be seen as dynamic (since code is executed at compile time to produce pieces of programs). In particular, this allows to lift most of the restriction imposed on multi-stage functors.

Acute (Sewell et al. 2007) is an extension of OCaml for distributed programming. It provides typesafe serialization and deserialization and also allows arbitrary loading of modules at runtime. Like Eliom, it provides a full-blown module system. However, it takes an opposite stance on the execution model: each actor runs independent programs and communications are completely dynamic. Handling of multiple type universes is done by providing a description of the type with each messages and by versioning APIs.

CONCLUSION

We presented Eliom$_m$, a tierless Web programming language with an expressive ML-like module system. Its expression language is a reformulation of Eliom$_\varepsilon$ (Radanne et al. 2016a). It features a module language with manifest types in the style of Leroy (1995) with base, client, server and mixed modules. Additionally, we were able to propose a limited notion of mixed functors that contain both client and server declarations. We gave a compilation scheme that transforms tierless programs in two non-tierless programs, one for the client and one for the server. We showed that our compilation scheme preserves the typing relations and provides good support for separate compilation.

The Eliom$_m$ language aims to model our current implementation of Eliom, an extension of the OCaml programming language with tierless annotations. This extension was designed to integrate well with the OCaml language which allows to use OCaml modules and compilation products from the vanilla OCaml compiler directly.

The need for a module system which integrates tierless annotations comes directly from the development of libraries and Web applications as part of Ocsigen. Web sites have become increasingly complex in the past decade. While several solutions for the "tiers" problem has been proposed, very few tackle the practical issues raised by programming large web applications with tierless languages. We believe that good support for modularity and abstraction is essential for any serious large-scale programming.

While the notion of locations and mixed modules achieve the desired modularity, several questions remain open regarding mixed functors. Functors are very expressive, combining this expressivity with a static slicing operation and a one-way communication scheme is challenging. We proposed to impose careful restrictions over the shape of mixed functors that allow to slice them statically, associated with some code transformations that help the slicing operation. We believe these restrictions can be further relaxed. User feedback should give us insight on how mixed functors are used and which kind of programming idioms and library organization they allow.

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. 2014. Rethinking Traditional Web Interaction: Theory and Implementation. *International Journal on Advances in Internet Technology* (2014). http://www.iariajournals.org/internet_technology/

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.

James Cheney, Sam Lindley, Gabriel Radanne, and Philip Wadler. 2013. Effective Quotation. *CoRR* abs/1310.4780 (2013). http://arxiv.org/abs/1310.4780

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.

Karl Crary. 2017. Modules, abstraction, and parametric polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 100–113. http://dl.acm.org/citation.cfm?id=3009892

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.

Scott Kilpatrick, Derek Dreyer, Simon L. Peyton Jones, and Simon Marlow. 2014. Backpack: retrofitting Haskell with interfaces. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 19–32. DOI: http://dx.doi.org/10.1145/2535838.2535884

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

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 109–122. DOI: http://dx.doi.org/10.1145/174675.176926

Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 142–153. DOI: http://dx.doi.org/10.1145/199448.199476

David B. MacQueen. 1984. Modules for Standard ML. In *LISP and Functional Programming*. 198–207.

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

MLton 2014. MLton. (2014). http://mlton.org/Home

Olivier Nicole. 2016. Bringing typed, modular macros to OCaml. (2016). https://oliviernicole.github.io/about_macros.html

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

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. 2016a. 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

Gabriel Radanne, Jérôme Vouillon, Vincent Balat, and Vasilis Papavasileiou. 2016b. Eliom: tierless Web programming from the ground up. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL*. ACM.

Claudio V. Russo. 2000. First-Class Structures for Standard ML. *Nord. J. Comput.* 7, 4 (2000), 348–374.

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.

Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2007. Acute: High-level programming language design for distributed computation. *J. Funct. Program.* 17, 4-5 (2007), 547–612. DOI: http://dx.doi.org/10.1017/S0956796807006442

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. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014. (EPTCS)*, Oleg Kiselyov and Jacques Garrigue (Eds.), Vol. 198. 22–63. DOI: http://dx.doi.org/10.4204/EPTCS.198.2

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94.

Jeremy Yallop and Leo White. 2015. Modular macros. *OCaml Workshop* (2015). http://www.lpw25.net/ocaml2015-abs1.pdf